# BALANCING GENERALITY AND SPECIALIZATION FOR MACHINE LEARNING IN THE POST-ISA ERA

A Dissertation
Presented to
The Academic Faculty

By

Divya Mahajan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2019

# BALANCING GENERALITY AND SPECIALIZATION FOR MACHINE LEARNING IN THE POST-ISA ERA

Approved by:

Dr. Hadi Esmaeilzadeh, Advisor
Department of Computer Science and Engineering
*University of California, San Diego*

Dr. Doug Burger
*Microsoft Corporation*

Dr. Hyesoon Kim
School of Computer Science
*Georgia Institute of Technology*

Dr. Milos Prvulovic
School of Computer Science
*Georgia Institute of Technology*

Dr. Dean Tullsen
Department of Computer Science and Engineering
*University of California, San Diego*

Date Approved: March 15, 2019

Somewhere, something incredible is waiting to be known!

*Carl Sagan*

*To Mumma and Papa!*

## ACKNOWLEDGMENTS

This thesis is dedicated to my parents, Dr. Amita Mahajan and Dr. Anil Kumar, who have always shown unbelievable faith in me. They were my first teachers, who taught me to believe in science and dedicate my career for its advancement. Even though they might not understand most of this thesis, this dissertation would be meaningless without them.

Everything I have achieved in the past five years of my PhD would not have been possible without the wholehearted support and dedication of my advisor, Dr. Hadi Esmaeilzadeh. He has not only taught me how to do high-quality and impactful research but has been a terrific mentor at every step of this doctoral journey. I am forever indebted to him for spending ample amounts of his time to coach us on how to write research papers, develop ideas, and conduct structured research. Hadi cares for his students like his own and I look forward to continuing this long-lasting relationship with him as – an advisor, a friend, and family.

I would also like to thank my committee members, Dr. Doug Burger, Dr. Hyesoon Kim, Dr. Milos Prvulovic, and Dr. Dean Tullsen for their constant encouragement, guidance, and insightful comments. Particularly, I would like to extend my gratitude to Doug, who has always offered alternative perspectives towards my research and enabled me to focus on fundamental concepts and novelties. I admire his astute insights and look forward to learning from him at every step of my career. A special thanks to Hyesoon for always being there; in addition to her support, she gave me deep technical comments, posed insightful questions, and provided invaluable feedback to improve the way I present my work.

My colleagues in Alternative Computing Technologies (ACT) lab, Joon Kyung Kim, Jongse Park, Jacob Sacks, Hardik Sharma, Amir Yazdanbakhsh, Emmanuel Amaro, and Sean Kinzer, have played a critical role in the success of this PhD. Joon has spent numerous hours during his undergraduate and masters years, helping me with my projects, even though it was not required of him. He provided me with all the support to complete the

projects in a timely and comprehensive manner. Many thanks to Jongse, Hardik, and Sean for the stimulating technical discussions, and Amir and Emmanuel for all the assistance on the projects. To Jake for thoroughly proofreading all my papers and helping me overcome all the rough patches during this journey.

I would also like to extend my gratitude to the faculty in College of Computing at Georgia Tech who are always there to guide me, especially after Hadi moved to UC, San Diego. They made me feel included and cared for, despite not being one of their students. A special thanks to Dr. H. Venkateswaran, who made the move from Austin and the entire PhD process very smooth. To Dr. Vijay Janapa Reddi and Dr. Annie Anton for being great mentors. To Dr. Derek Chiou, who taught me my first computer architecture class and kindled my interest in this field. I owe it to the faculty at UT Austin for teaching me the fundamentals of computer architecture and design. Thanks to my collaborators Arun Kumar and Adel Ardalan; Arun for providing in-depth insights into databases which were critical for success of our projects and Adel for enabling me to take a step back and look at the bigger picture.

My gratitude to Eric Chung, Adrian Caulfield, and Jeremy Fowers for providing me the opportunity to work in the Catapult team at Microsoft Research. Many thanks to Joel Emer and Angshuman Parashar for offering me an internship at Nvidia. These experiences gave me the opportunity to interact with brilliant people, work on cutting edge research, and make lifelong relationships.

I would like to extend my profound gratitude to my entire family in Chandigarh, India, who have taught me the true meaning of unconditional love. To my grandparents, Smt. Usha Kumari, Shri. Pratap Chand Gupta, Smt. Raj Mahajan, and Shri. Rajpal Mahajan, who have inculcated in us the importance of education; my in laws, Mrs. Seetha Sekar and Mr. Chandrasekaran Natarajan, who have showered me with blessings; my sister-in-law Vinita and brother Abhishek for all their love; my uncles Jatinder and Sandeep, aunts Parveen and Poonam, cousins Anubha, Akshit, and Anchit, who have always been by my

side; my sister-in-law Aparna and brother-in-law Rathnam for their continuous encouragement. And finally to my friends and family in the US, an incomplete list includes Prashant, Anshika, Parminder, Divya, and Michael, who are constantly enriching my life.

Last but never the least, I would like to thank my husband Balaji Chandrasekaran, without whom this PhD would not have been possible. His patience, perseverance, and dedication towards my growth has been unwavering, even during my emotional absences and tough times. He has spent numerous hours proofreading and critiquing my work even though he did not understand most of the technical content. He has been there by my side during all the deadlines, late work hours, conference presentations, and internships, even if it meant sleeping in the lab or bringing coffee at odd hours. Balaji you are a true partner; words cannot even begin to express my gratitude towards you.

Divya Mahajan
Atlanta, Georgia

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# Balancing Generality and Specialization for Machine Learning in the Post-ISA Era

**Abstract.** A growing number of commercial and enterprise systems are increasingly relying on compute-intensive machine learning algorithms. While the demand for these applications is growing, the performance benefits from general-purpose platforms is diminishing. This challenge has coincided with the explosion of data where the rate of data generation has reached an overwhelming level that is beyond the capabilities of current computing systems. Therefore, applications such as machine learning and robotics can benefit from hardware acceleration. Traditionally, to accelerate a set of workloads, we profile the code optimized for CPUs and offload the hot functions on hardware compute units designed specially for that particular function, hence providing higher performance and energy efficiency. Instead in this work, we take a revolutionary approach where we delve into the algorithmic properties of applications to define domain-generic hardware acceleration solutions. We leverage the property that a wide range of machine learning algorithms can be modeled as stochastic optimization problems. Using this insight we devise compute stacks for hardware acceleration that are built independent of the CPU. These stacks expose a high-level mathematical programming interface and automatically generate accelerators for users who have limited knowledge about hardware design, but can benefit from large performance and efficiency gains for their programs.

Keeping these ambitious goals in mind, our work (1) strikes a balance between generality and specialization by breaking the long-held traditional abstraction of the Instruction Set Architecture (ISA) in favor of a more algorithm-centric approach; (2) develops hardware acceleration frameworks by co-designing a language, compiler, runtime system, and hardware to provide high performance and efficiency, in addition to flexibility and programmability; (3) segregates algorithmic specification from implementation to shield the

programmer from continual hardware/software modifications while allowing them to benefit from the emerging heterogeneity of modern compute platforms; and (4) develops real cross-stack prototypes to evaluate these innovative solutions in a real-world setting and make them open-source to maximize community engagement and industry impact. Our work TABLA (`http://act-lab.org/artifacts/tabla/`) is public, and defines the very first open-source hardware platform for machine learning and artificial intelligence.

***Thesis statement**: Conventionally, to accelerate a set of workloads, we identify a compute-intensive function within a program that is generated for the Instruction Set Architecture (ISA). We offload the function onto specifically designed compute units to obtain higher performance and energy efficiency. Instead, in this thesis, we take an alternative approach towards hardware acceleration by focusing on identifying algorithmic commonalities across emerging intelligent applications. We devise domain generic full-compute stack solutions by leveraging the insight that a wide range of machine learning algorithms can be modeled as stochastic optimization problems. In this dissertation, we re-think the hardware-software abstraction and raise it to the algorithmic level in lieu of the canonical ISA. Our solutions allow programmers to express their applications intuitively via a high-level programming interface. User programs are then automatically mapped to hardware accelerators that are specialized for compute sequences, data transformations, and data read/write patterns commonly seen in our target domain.*

# SUMMARY

Training a machine learning model requires ample amounts of computation that is repeatedly applied over the training data for large number of iterations. While the demand for these computationally intensive algorithms is increasing, the benefits from general-purpose computing are diminishing [1, 2, 3]. As a result, both the industry [4, 5, 6] and the research community [7, 8, 9, 10, 11] are increasingly focusing on hardware accelerators, which can provide large gains in efficiency and performance by devising specialized frameworks for restricted workloads. Traditionally, creating acceleration solutions require expertise in the application domain and hardware design to specify, synthesize, and run processors that are highly specialized and efficient for compute intensive algorithms such as digital signal processing, machine learning, genomics, and many more. Instead, in this thesis, we delve into the algorithmic properties of the application to design a comprehensive full stack solution from programming language down to circuits, where each layer of the stack is designed by keeping in mind the the properties of the algorithm. These full stack solutions can generate hardware accelerators without any manual intervention or requirement of hardware expertise from the user.

**A Unified Template-Based Framework for Accelerating Classical Machine Learning.** Our foremost work on the concept of using algorithmic properties to design a full-stack solution is TABLA [12], which leverages the insight that the training of a wide range of supervised machine learning algorithms can often be modeled as a stochastic optimization. Such machine learning algorithms can be optimized using stochastic gradient descent which iterates over the training data, minimizes a loss function, and updates the parameters that capture the patterns in the data. For TABLA, stochastic gradient descent forms the abstraction between the hardware and software to resolve two conflicting objectives – automation and high performance. To ensure automation, this framework exposes a domain specific language for the user to specify the algorithm, which is then converted into the final

accelerator by TABLA's compiler and design builder. To obtain high performance, the hardware backend is implemented as a hand-optimized template-based architecture comprising the general framework of stochastic gradient-based optimization. TABLA's compiler and design builder tailor this template design to generate a hardware accelerator specifically for each algorithm that delivers high-performance and efficiency. TABLA automatically customizes these templates according to the {learning algorithm, FPGA} pair and generates synthesizable Verilog code. TABLA has been described in detail in Chapter 2 and is the first line of this thesis work; it targets the machine learning and data analytics part of the data processing pipeline shown in Figure 1.1.

**Integrating hardware acceleration within the software stack of data management systems.** In addition to data analysis, data management and retrieval, especially from a complex Relational DataBase Management System (RDBMS) is a crucial component of the horizontal data processing pipeline. Past work has exclusively focused on either accelerating machine learning or data retrieval/querying independently but has inadvertently ignored the entire horizontal pipeline of systematic data processing, which entangles both retrieval and learning. Leveraging only the properties of either one generates a suboptimal solution which does not mitigate the unique bottlenecks resulting from the interplay between different components of this pipeline. To tackle this, we devise DAnA [13], a solution that merges three disjoint research areas of this pipeline (enterprise in-database analytics, modern acceleration platforms, and analytical programming paradigms) to enable transparent and efficient hardware acceleration for in-RDBMS advanced analytics. The machine learning training algorithm is provided by the user and expresses how each record/tuple in the training data table updates its model, how results from multiple tuples can be combined, and when the algorithm terminates. DAnA then automatically generates a hardware accelerator design suitable for this high-level specification provided by the user. Our hardware design circumvents the CPU from the data retrieval process by integrating a hardware

component directly with the RDBMS engine to read and process the training data table obtained from the buffer pool of the database. The processed training data is then sent to the part of the chip that accelerators the learning algorithm. Data scientists with no hardware design expertise can use DAnA to harness acceleration without manual data retrieval and extraction. DAnA is discussed in detail in Chapter 3.

**Enabling Methodical and Controlled Approximation for High Performance Hardware Designs.** Solutions such as TABLA and DAnA utilize a template-based approach where the templates are designed by expert hardware designers for the target domain but offer reconfigurability for variety of algorithms. Both the full-stack solutions briefly described above automatically reconfigure the template in accordance to the requirements of the specified algorithm and resource availability on the hardware platform. Although, acceleration and specialization in itself provides higher performance and efficiency in comparison to general purpose compute systems, these templates can further benefit from approximation. The first milestone in this overarching area of research involved devising *Axilog*– a set of concise, intuitive, and high-level annotations that provide the necessary syntax and semantics for approximate hardware design and reuse in Verilog [14, 15]. Axilog enables designers to delineate which parts of a hardware system or circuit design are critical and cannot be approximated. A key factor in our language formalism is to abstract away the details of approximation while maintaining the designer's oversight in deciding which circuit elements are synthesized approximately. Axilog is also devised with modular reusability as a first order consideration. This is because hardware system implementations rely on modular design practices where the engineers build libraries of modules and reuse them to build more complex hardware systems.

In addition to facilitating approximation, there is a need to explore quality control mechanisms. For wide-scale acceptability of imprecise results, we need mechanisms that can control the degree of approximation. Thus, another line of our work defines a cohesively

co-designed hardware-software solution, MITHRA [16], with components in both compiler and microarchitecture to ensure quality control for approximate accelerators. The common practice in approximate acceleration is to *always* invoke the accelerator in lieu of a frequently-executed safe-to-approximate region of code, e.g., a function in a loop. Always invoking the accelerator provides maximum gains from approximation but can potentially lead to an unacceptable *fixed degree* of quality loss. This approach does not provide the flexibility to explore the tradeoffs between quality and gains in performance and efficiency. MITHRA seeks to identify whether each individual accelerator invocation will lead to an undesirable quality loss. If MITHRA speculates that a large quality degradation is likely, it directs the processor to run the original precise code. This solution provides a knob to the software to control the quality tradeoffs and solves a statistical optimization problem to tune the knob. MITHRA provides statistical guarantees with a high confidence that desired quality loss levels will be met on unseen datasets. Both Axilog and MITHRA are described in Chapter 4.

# Thesis Contributions

- **Unifying accelerators: An algorithmic approach toward acceleration.** One of the most important aspects of this thesis is unification of accelerator design for a wide range of machine learning algorithms based on their fundamental algorithmic properties. While a large body of work aims to devise accelerators for different machine learning algorithms, the common practice of accelerator design relies mostly on identifying hot regions of code through profiling a set of implementations. This ties the accelerators to a certain implementation and limits their applicability. We dive deep in the theory of machine learning and provide an alternative way of devising accelerator-based systems by finding a common umbrella for these disparate efforts. This unification, however, does not prevent our frameworks to tailor-make a hardware accelerator for each algorithm to deliver high-performance and efficiency by specializing its templates. We accomplish this by using a template-based paradigm for accelerator design that combines hand-optimized templates and domain-specific languages to achieve both efficiency and generality, respectively. Delivering on these conflicting fronts is of utmost importance for mainstream adoption of acceleration.
- **Rethinking the abstraction between hardware and software.** For decades, the ISA have served as the canonical abstraction between hardware and software. Specialization and acceleration breaks this abstraction. By identifying theoretical commonalities, we provide a novel approach in defining the abstraction between hardware and software. This approach is apropos, since the community is rethinking the traditional abstractions to not only deliver higher performance and better efficiency but maintain some degree of generality. Our abstraction dissociates hardware acceleration from specific software programs that are mostly targeted for CPU execution, hence are constrained by the implementation's programming language.

- **Bridging different paradigms.** The community has exclusively focused on either accelerating machine learning or data retrieval/querying but has inadvertently ignored the entire horizontal pipeline of systematic data processing, which entangles both learning algorithms and data retrieval. Leveraging only the properties of the machine learning algorithm generates a suboptimal solution that does not mitigate the unique bottlenecks resulting from the cross-pollination of different components in this pipeline. This thesis takes a first step towards understanding this horizontal stack of data processing and devises a vertical full-stack solution, from programming languages to template-based hardware, by addressing the challenges observed in executing advanced analytics within an RDBMS setting. To this end, unifying these research directions helps mitigate the inefficiencies and reduced productivity of data scientists by enabling them to benefit from in-database hardware acceleration for analytics whilst retaining familiar programming environments.
- **Statistical quality guarantees in approximate computing.** Specialization and approximation are a means to provide high performance and energy efficiency to overcome the shortcomings of the benefits being obtained from general purpose systems. We found that training machine learning with specialized hardware opens avenues to run compute intensive algorithms which otherwise take a significantly longer time to finish. However, these algorithms are inherently amenable to approximation, and it is clear that the applicability of approximate computing requires moving beyond traditional and formal quality guarantees. Our work takes an initial step in controlling the quality tradeoffs for approximate accelerators; aiming to open a path for their adoption. This thesis provides statistical quality guarantees that user-defined quality requirements will be met with approximate accelerators on unseen data.

# Thesis Organization

This thesis is organized as follows; Chapter 1 provides a high-level overview of the already existing trends in the computing industry especially for data management and analysis. The chapter goes into the details of some of these trends such as: (1) algorithmic properties that are common across a wide range of machine learning algorithms, hence, form the fundamentals of our comprehensive full stack solutions and (2) how these algorithms integrate within the current relational database management systems, i.e., in-database analytics. Finally, the chapter also provides a brief overview of Field programmable Gate Arrays (FPGAs), our prototyping platform for hardware accelerators. Chapter 2 then delves deeply into our first work TABLA and provides details about how we use algorithmic properties of certain supervised machine learning algorithms to devise a comprehensive full stack solution. The chapter describes different components of this full stack solution such as Domain Specific Language, Model Compiler, Design Builder, and Template-Architectures, and how they are stitched together to automatically generate FPGA-based accelerators. Chapter 3 details the integration of hardware acceleration within Relational Database Management Systems (RDBMs). It discusses how the entire hardware acceleration full stack works in tandem with the RDBMS to execute machine learning training on an FPGA when the data is stored within the realms of a database table. The goal is to avoid exporting the training data out of the database, circumvent the CPU from the data transformation process, and send it to the machine learning accelerator to perform the analytics.

Chapter 4 delves into the approximate computing component of this thesis, where it first discusses the language extensions for Verilog that enable a hardware designer to express which part of their hardware design can be approximated. The chapter then further dives into our quality control mechanism for approximate accelerators, i.e., how we devise a hardware-software co-design that exposes knobs to the software that can be tuned to control the quality tradeoff during runtime. Chapter 5 then provides a brief overview of other works

that the author has contributed to in collaboration with her colleagues. These works lie in both the acceleration for machine learning and approximate computing domain. Finally, the Chapter 6 provides some of the potential future directions which spawn from this thesis.

# CHAPTER 1

# A NEW ERA IN COMPUTING

Data is increasing at an unprecedented rate. There is a growing need to efficiently collect, process, and analyze this large influx of data. A broad range of industries such as finance, manufacturing, retail, social networking, and e-commerce are using databases to organize their data. Post collection, every industry is keen on processing this data to employ profit increasing techniques such as targeted advertising, dynamic pricing, and demand forecasting. These data analysis applications often rely on machine learning to accomplish their objectives. In fact, the advances in machine learning are changing the landscape of computing towards a more personalized and targeted experience for users. However, these machine learning algorithms are computationally intensive workloads. Furthermore, with the effective end of Dennard scaling [3], traditional CMOS scaling no longer provides performance and efficiency gains commensurate with increases in transistor density [1, 2, 8]. The current paradigm of general-purpose processor design falls significantly short of the traditional cadence of performance improvements [17]. As such, companies and research communities are increasingly moving towards specialized hardware accelerators that provide orders of magnitude higher performance. While specialized hardware shows much promise, they are often difficult to integrate within the existing high-level software stacks commonly used by programmers and data analysts. In this chapter, we discuss some of the challenges prevalent in the data processing landscape. We also provide a brief overview of machine learning and DBMSs that can enable us to mitigate some of these challenges.

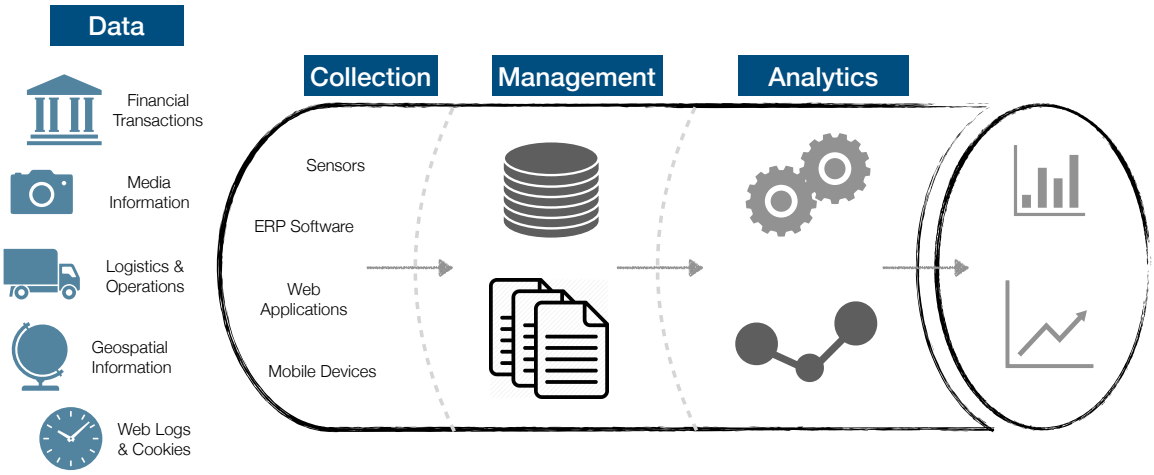



**Figure 1.1:** A visualization of the data processing pipeline. Due to ubiquity and pervasiveness of the internet, data is being generated by numerous devices. This data is then managed through different means ranging from databases to file systems. Then in the data analysis phase, machine learning algorithms are commonly applied on this data to extract insights. These insights are then used to predict future trends.

## 1.1 Data Processing Pipeline

The data processing pipeline shown in Figure 1.1 depicts flow of data, from collection, management, and analysis to prediction. Huge amounts of data is being collected from smart devices, such as sensors, social media, cookies in web applications and much more. Since the advent of the internet and due to the continuous growth in the compute and memory resources, there has been an exponential increase in data generation that can be stored for future analysis. This data is being collected across domains such as financial transactions, media information such as images and videos, geospatial data such as location coordinates, etc. Conventionally, data has been stored in database management systems as tables, but now a lot of the information is being generated in unstructured formats. Thus, unconventional management mechanisms such as geospatial databases, file formats like JSON and XML are gaining traction. Usually, most of the management techniques offer either integrated or external analysis tools. These tools often employ machine learning algorithms or statistical techniques to find patterns and insights from the data. Finally, these patterns are then used to predict future trends. The utilization of various types of devices

to generate data, management of this information, and complex computations required for analysis and machine learning on this data demands new technological approaches. The goal is to bridge the gap between data growth and performance improvements from general purpose compute. CPUs are no longer providing the required performance improvements, hence there has been an increasing interest in coalescing high-performance specialized hardware architectures within the traditional CPU ecosystem.

Designing efficient accelerators and harnessing the available compute resources on current platforms requires extensive expertise in both hardware design and the application domain. Integrating hardware acceleration and heterogeneous compute in the data processing pipeline, shown in Figure 1.1, is further exacerbated due to the complex cross-pollination of data collection, data management, machine learning, and prediction. In this thesis, we aim to tackle the intertwined challenges for a part of this data processing pipeline which is, integrating FPGAs to accelerate the training phase of machine learning within Relational Database Management Systems (RDBMS) – by taking a holistic approach that reworks the fundamental hardware-software abstractions.

Traditionally, to design such accelerators for a set of workloads or an application domain, we profile the code that is generated for the traditional Instruction Set Architecture. If the code contains a compute intensive function, we offload it onto hardware compute units that are designed specifically to run this function, hence obtaining higher performance and energy-efficiency. Instead, we take a revolutionary approach where the goal is to understand the algorithmic properties of a domain and create hardware acceleration solutions that incorporate those properties independent of the CPU. Therefore, our hardware acceleration solutions are domain generic, where the entire computing stack is devised and optimized for a wide range of algorithms. The target application for this thesis is supervised machine learning and we find that such algorithms can be modeled as stochastic optimization problems. As the training phase of these algorithms is very compute intensive, hardware acceleration offers a potent solution to overcome the gap between continuously increasing

data and diminishing returns from performance improvement of general purpose compute systems. In the next section, we discuss the algorithmic properties of the training process of supervised machine learning algorithms.

## 1.2 Training in Machine Learning

In the data processing pipeline, a wide range of data analysis is performed using machine learning algorithms. Supervised machine learning algorithms generally involve two phases: the training phase and the prediction phase. The training phase, which is precursory to the prediction phase, generates a model that maps one or more inputs (independent variables) onto one or more outputs (dependent variables). The generated model is used in the prediction phase to predict the dependent variables for new unseen inputs. The training phase is more compute intensive and can benefit significantly from acceleration as in this phase a wide range of algorithms go through a cyclic process of constant iteration, tuning, and improvement.

Each machine learning algorithm has a specific loss function that captures the measure of error between the mapping of the independent variables onto the dependent variable (X$\rightarrow$Y i.e $Y = h(W, X)$) and the golden dependent variable ($Y^*$). Here X are the independent variables, $Y$ is the dependent variable, and $W$ are the model parameters that the training process aims to find, where the final $W$ minimizes the difference between golden $Y^*$ and predicted Y. Improving the model corresponds to minimizing this loss function using an *optimization algorithm*, which is applied repeatedly over the training model until convergence. Next, we discuss a learning algorithms can be expressed as stochastic optimization problems [18]. Examples of such learning algorithms are support vector machines, logistic regression, least square models, backpropagation, conditional random fields, recommender systems, Kalman filters, linear and nonlinear regression models, and softmax functions.

### 1.2.1 Learning as an Optimization Problem

For certain machine learning algorithms the learning task becomes solving an optimization using an optimizer such as stochastic gradient descent [19] or conjugate gradient that iterates over the training data and minimizes a loss function. Although the loss function varies for different learning algorithms, the optimizer or the solver is fixed. Therefore, the accelerator for these learning tasks can be implemented as a template design, uniform across a class of machine learning algorithms as discussed in Chapter 2. This template design comprises the general framework for the optimizer, one of the most common optimizers being gradient descent methods. Next we give a brief overview of these methods.

#### *1.2.1.1 Gradient Descent Methods*

As aforementioned, each machine learning algorithm in our target class is distinguished by its loss or objective function. The objective function has a set of parameters that are learned in accordance with the training data such that the learned model can make data-driven predictions or decisions on new unseen data. During each iteration, the objective function quantifies the error between the current model's output or dependent variable (prediction) and the expected output given with the training data. Thus, a machine learning algorithm learns a model by solving an optimization problem that minimizes the prediction error over the entire training data as shown in Equation (1.1).

$$\min_{W^{(t)} \epsilon \mathbb{R}} \sum_{i} f(w^{(t)},\ x_i) \tag{1.1}$$

In Equation (1.1), $x_i$ is the $i$th input, $W^{(t)}$ is the model parameter at iteration $t$ and $f(W_i^{(t)}, x_i)$ is the prediction error. The prediction error function $f(W^{(t)},\ x_i)$ could be as simple as a square of the difference between the $i^th$ predicted output and known output, i.e., $(h(W^{(t)},\ x_i) - y_i^*)^2$. However, as mentioned this objective or loss function changes with the machine learning algorithm. Nonetheless, the sum of the prediction errors across

all training input vectors is the objective function that needs to be minimized. To learn a model $W$, optimization algorithms iterate over the training data and gradually reduce the prediction error by changing the model parameters. Gradient descent is one such common optimization solver. While the gradient descent algorithm is fixed across different machine learning algorithms, the objective function varies.

**Gradient descent.** The gradient descent algorithm starts with an initial set of model parameters and iteratively moves towards a set of parameters that minimize the objective function. This iterative minimization is achieved by taking steps in the decreasing direction of the objective function's derivative or gradient. Hence, for each iteration, the parameters $W^{(t)}$ are updated as shown below.

$$W^{(t+1)} = W^{(t)} - \mu \times \frac{\partial(\sum_i \ f(W^{(t)},\ x_i))}{\partial W^{(t)}} \tag{1.2}$$

As Equation (1.2) shows, $W^{(t+1)}$ is updated in the negative direction of the objective function's gradient ($\frac{\partial f}{\partial W^{(t)}}$) with learning rate, $\mu$. In a single iteration of gradient descent, the gradient of the objective function is calculated over the entire training data to obtain the next set of parameters $W^{(t+1)}$. This process is repeated until the function is minimized and the final set of parameters $W^{(final)}$ is obtained. $W^{(final)}$ is the trained model of the machine learning algorithm for a given training dataset. For very large training datasets, gradient descent can impose a high overhead as it calculates a sum over the entire data in a single iteration. To avoid this computationally large overhead, stochastic gradient descent is generally used [18, 19, 20].

**Stochastic gradient descent.** Stochastic gradient descent is a modification of the conventional gradient descent algorithm. It divides the objective function into smaller differentiable functions. As Equation 1.1 shows, the objective function is a summation of a function over the entire training data. Instead of taking the derivative of the function cal-

**Table 1.1:** Machine learning algorithms, their objective function, and the gradient of this objective function (used with **T**ABLA). The $\delta()$ operator in the objective and gradient functions represents a complex nonlinear transformation. For example, in logistic regression $\delta(W, X_i)$ is $sigmoid(\sum_i X_i \times W)$.

| Machine Learning Algorithm | Objective Function ($f$) | Gradient of the Objective Function ($\partial f$) |
|---|---|---|
| **Logistic Regression** | $\sum_i \{ Y_i \log(\delta(W, X_i)) + (1 - Y_i) \log(1 - \delta(W, X_i)) \} + \lambda\|\|W\|\|$ | $(\delta(W, X_i) - Y_i)X + \lambda W$ |
| **Support Vector Machines** | $\sum_i \{ 1 - Y_i W X_i \} + \lambda\|\|W\|\|$ | $Y_i X_i + \lambda W$ |
| **Recommender Systems** | $\sum_{i,j} \{ (Y_{ij} - W_j X_i)^2 \} + \lambda\|\|W, X\|\|$ | $\sum_i (Y_{ij} - W_j X_i)X_i + \lambda W$ , $\sum_j (Y_{ij} - W_j X_i)W_j + \lambda X$ |
| **Backpropagation** | $\sum_i \sum_k \{ (Y_i^k \log(\delta(W, X_i)^k)) + (1 - Y_i^k) \log(1 - \delta(W, X_i)^k) \} + \lambda\|\|W\|\|$ | $\sum_k (\alpha_i^k \Delta_i^k) + \lambda W$ |
| **Linear Regression** | $\sum_i \{ \frac{1}{2} (WX_i - Y_i)^2 \} + \lambda\|\|W\|\|$ | $(WX_i - Y_i)X_i + \lambda W$ |

culated over the entire dataset, stochastic gradient descent divides the objective function into smaller functions requiring a single input vector. Therefore, the gradient of the smaller function is only calculated over a single vector. Thus, the parameter update rule changes from Equation (1.2) to Equation (1.3).

$$W^{(t+1)} = W^{(t)} - \mu \times \frac{\partial f(W^{(t)},\ x_i)}{\partial W^{(t)}} \tag{1.3}$$

The calculation in Equation (1.3) is repeated individually for all input vectors $X_i$, until the function converges to its minimum value. Stochastic gradient descent typically takes more iterations to converge in comparison to conventional gradient descent. However, the benefits obtained by avoiding data accesses to all the input vectors for each iteration significantly outweigh the cost incurred by executing more iterations. Using stochastic gradient descent to find the minimum of the objective function is imperative for large training datasets across different domains of machine learning algorithms. A small subset of algorithms that can be optimized using stochastic gradient descent are provided in Table 1.1; the table also specifies each algorithm's objective function and their respective gradients. Algorithms such as conditional random fields, Kalman filters, portfolio optimization, least square models, logistic regression, SVM, recommender systems, back-propagation, and linear regression can be optimized using stochastic gradient descent.

The insight that many algorithms can be optimized using stochastic gradient descent motivated us to choose it as the abstraction between the software and the hardware for one

of the first works of this thesis, TABLA [12]. The key was to identify such commonalities across a wide range of machine learning algorithms and utilize this commonality to provide a high-level abstraction for programmers. The gradient descent solver is fixed while the objective function changes for different learning algorithms. TABLA provides a template-based framework to accelerate this class of learning algorithms. Therefore, a developer can specify the learning task by *only* expressing the gradient of the objective function using our high-level language. TABLA then automatically generates the synthesizable implementation of the accelerator for FPGA realization using a set of hand-optimized templates. We discuss the details of this work in Chapter 2.

#### *1.2.1.2 Iterative Optimization and Update Rules*

As discussed earlier, learning algorithms use optimization procedures that iteratively minimize their loss function – distinct for each learning algorithm – by using one input-output pair (tuple) at a time to generate updates for the model. Each machine learning algorithm has a specific loss function that mathematically captures the measure of the learning model's error. However, we have previously discussed the use of stochastic gradient descent as the optimizer to minimize the loss function. Instead of restricting the user to stochastic gradient descent, in the second work DAnA [13] (detailed in Chapter 3) we enable the user to specify their entire algorithm – both loss function and the optimizer. Therefore, in DAnA, improving the model corresponds to minimizing the loss function as per an *update rule*, which is applied repeatedly over the training model. The update rule in this case is the computation which specifies how a single training data tuple will update the machine learning model, an example of which is provided below.

**Example.** Given a set of $N$ pairs of $\{(x_1, y_1^*), ..., (x_N, y_N^*)\}$ constituting the training data, the goal is to find a hypothesis function $h(w^{(t)}, x)$ that can accurately map $x \rightarrow y$. The equation below specifies an entire update rule, where $l(W^{(t)}, x_i, y_i^*)$ is the loss function that

signifies the error between the output $y^*$ and the predicted output estimated by a hypothesis function $h(W^{(t)}, x_i)$ for input $x$.

$$W^{(t+1)} = W^{(t)} - \mu \times \frac{\partial(l(W^{(t)}, x_i, y_i^*))}{\partial w^{(t)}} \tag{1.4}$$

For each $(x, y^*)$ pair, the goal is to a find a model $(W)$ that minimizes the loss function $l(W^{(t)}, x_i, y_i)$ using an iterative update rule. The hypothesis function can vary greatly across different algorithms. It can range from being a linear or a non-linear equation to more complicated mathematical transformations such as a step function. As such, the two required components are the hypothesis function to define the machine learning algorithm and an optimization algorithm that iteratively applies the update rule. We provide the freedom to the user to express any type of these two functions as the entire update rule.

**Amortizing the cost of data accesses by parallelizing the optimization.** In Equation (1.4), a single ($x_i$, $y_i^*$) tuple is used to update the model. However, we can use a batch of tuples and compute multiple updates independently when the optimizer supports combining partial updates [21, 22, 23, 24, 25, 26, 27]. Examples of commonly used iterative optimization algorithms that support parallel iterations are variants of gradient descent and conjugate gradient, both of which can be applied across a diverse range of machine learning models. Using multiple tuples in an algorithm provides ample opportunities to the hardware accelerator to improve the utilization of its compute resources.

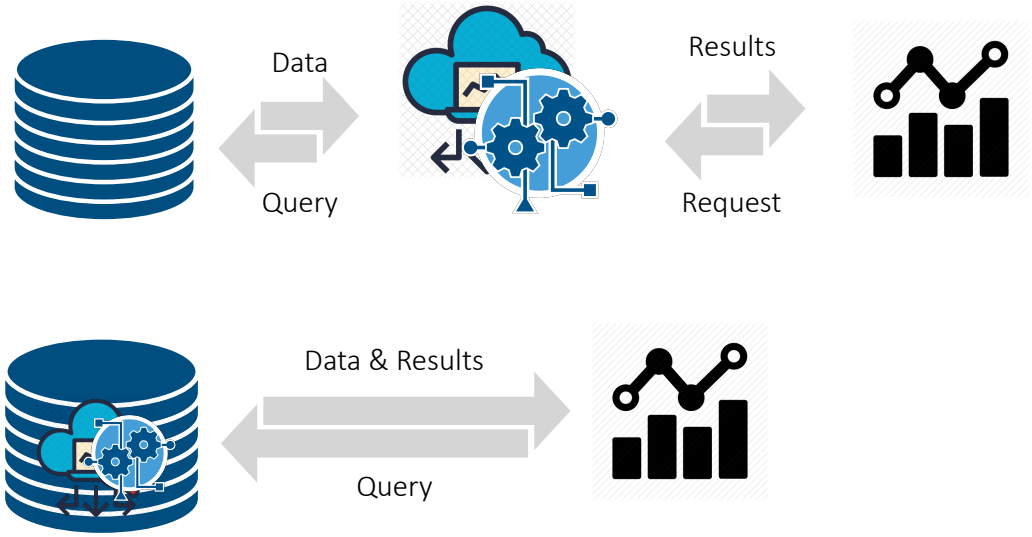
In the previous two sections we have discussed in detail the training process of supervised machine learning algorithms, i.e, data analysis of the processing pipeline shown in Figure 1.1. However, data management is another crucial aspect of this pipeline, hence, the database industry is investing in the integration of machine learning algorithms within RDBMSs, both on-premise and cloud-based [28, 29]. Next we discuss how data analytics and machine learning algorithms are integrated within the current software stack of data management. We then leverage these properties to inculcate hardware acceleration for

machine learning within database management systems (described in Chapter 3).

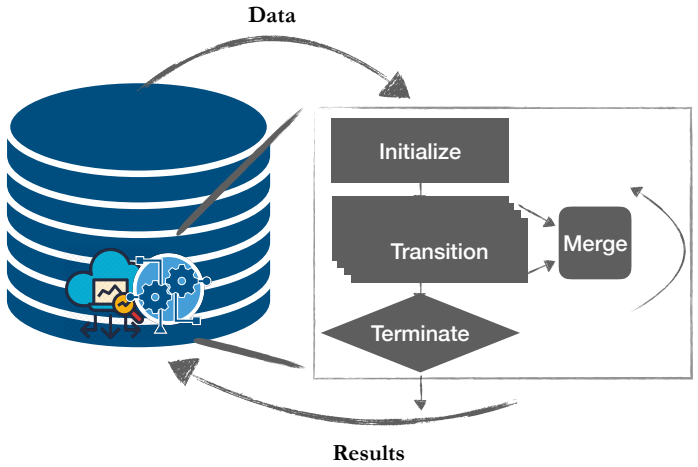
### 1.2.2 In-Database Analytics

Relational Database Management Systems (RDBMSs) are the cornerstone of large-scale data management in almost all major enterprise settings. However, data-driven applications in such environments are increasingly migrating from simple SQL queries towards advanced analytics, especially machine learning over large datasets [30, 31]. The database industry is investing in the integration of machine learning algorithms within RDBMSs, both on-premise and cloud-based [28, 29]. This integration enables enterprises to exploit machine learning without sacrificing the auxiliary benefits of an RDBMS, such as transparent scalability, access control, security, and integration with their business intelligence interfaces [32, 33, 34, 35, 36, 19, 37, 38, 39]. In-database analytics allows the data processing to be performed within a database, i.e., the analytical logic is built into a database instead of a separate application. This eliminates the overhead of moving large data sets out of the database and into a format that is suitable for the analytics applications provided for C/C++, R, and Python through external libraries. The comparison between an out of database and in-database approach is shown in Figure 1.2a. Advantages of in-database analytics includes parallel processing, scalability, and analytics optimization.

Database engines allow the user to specify User Defined Functions (UDFs) to express complex analyses on data stored in tables. Each database supports different languages to write these UDFs, including C/C++, Python, R etc., however, for machine learning training these functions follow a logical structure shown in Figure 1.2b. The *initialize* function allows the user to provide an initial state of the machine learning model. The *transition* function provided by the user is a set of operations that specify how a single tuple/record in the training data table updates the learning model. If multiple transition functions are spawned to consume multiple tuples, the *merge* function provides the means to combine all the updates to the model. Finally, the *terminate* function provides the convergence




**(a)** Comparing the conventional out of database analytics and in-database analytics. In the conventional approach the data is exported out the database and analytics is run on it using external libraries in Python or R. The in-database analytics approach removes the overhead of data export. It instead allows the database allows users to write complex analysis functions in languages such as C, C++, python etc., which are performed directly on the data stored in the database format.




**(b)** A structure of the analytics core as a User Defined Function.

**Figure 1.2:** In-Database Analytics. (a) Compares in-database and an out of database approach. (b) Specifies a structure for User Defined Functions that express the analysis that needs to be performed on data store in the database.

criterion. In DAnA [13] (described in Chapter 3) data scientists with no hardware design expertise can run such in-database analytics on hardware accelerators and harness their high-performance without manual data retrieval and extraction whilst retaining familiar programming environments.

As mentioned before, the training phase of machine learning can benefit from hardware acceleration. Accelerators can be designed for various platforms, such as Field Programmable Gate Arrays (FPGAs), Graphical Processing Units (GPUs), Application Specific ICs (ASICs), Coarse-Grain Reconfigurable Arrays (CGRAs) and much more. FPGAs, however, provide a promising path forward to accommodate the needs of machine learning algorithms and represent an intermediate point between the efficiency of ASICs and the programmability of general-purpose processors. Before delving into the details of the different works of this thesis, we provide a brief overview of FPGAs as our prototyping platform.

## 1.3 Field Programmable Gate Arrays

This section provides an overview of Field Programmable Gate Arrays (FPGAs). As machine learning is a constantly evolving field, we use FPGAs to prototype our accelerators due to their reconfigurability and reprogrammability. They offer a potent solution for hardware acceleration as they can be re-synthesized to execute new learning-based algorithms. FPGAs can readily support advanced analytics, all while offering higher performance than CPUs and improved power-efficiency over GPUs.

An FPGA is an integrated circuit that comprises reprogrammable Look Up Tables (LUTs), which mimic digital logic by storing a corresponding design configuration in the Static Random Access Memories (SRAMs). Figure 1.3 illustrates a small portion of the reprogrammable logic blocks and specialized hardware available on an FPGA chip. Contemporary FPGAs also include hardened memories called Block RAMs (BRAMs) and Digital Signal Processing (DSP) cores, as on-chip local memory accesses and arithmetic units are

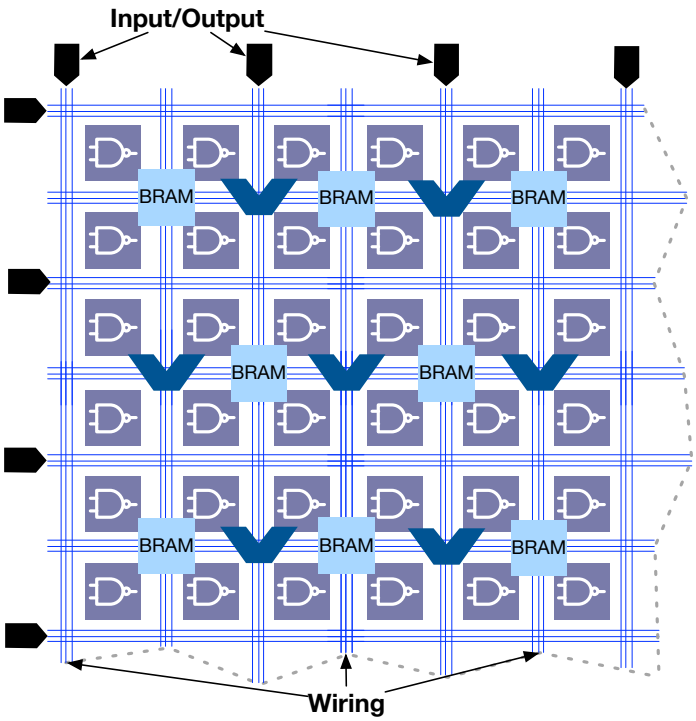



**Figure 1.3:** Illustration of a small corner of an FPGA chip. Modern FPGAs comprise a sea of binary lookup tables, augmented with hardened Block RAM and DSP slices to offer higher frequencies.

fundamentally required for many applications. Although FPGAs are energy efficient and reconfigurable, programming them is still a challenge and requires long design cycles, even for experts.

Hardware Description Languages (HDLs), such as Verilog and VHDL, provide a means to specify hardware logic at the register-transfer level (RTL). The designer is expected to provide a synthesizable code, i.e., a circuit description valid for the given FPGA. A typical hardware design flow demands iterative refinement of this description to verify its functional correctness. This verification process is a laborious task that involves rigorous simulations and cycle-by-cycle waveform generation to verify the functionality of the logic. Post verification, the programmer often has to optimize the design manually at the register or gate level. To reduce the complexity of programming with HDLs, companies offer proprietary high-level synthesis tools that convert C/C++ programs to Verilog or VHDL. The designer is still expected to go through the verification and optimization processes.

This thesis aims to exploit the reconfigurability and high performance of FPGAs for machine learning and advanced analytics without requiring the user to endure this painful design flow. Thus, we provide parametric architectures ($\sim$10,000–15,000 lines of Verilog code) designed *once* by hardware experts . These architectures can be tailored for each machine learning algorithm and target FPGA and are automatically configured by our full stack solution according to the application. We do so by delving into the algorithmic properties of the target machine learning algorithms and use them to specialize our novel full compute stacks that can reconfigure the parametric design. Furthermore, our solutions can be adapted for other platforms by modifying the template architectures and the backend of the stack.

We only require the analyst to provide the algorithm specification through a high-level programming interface. In our experience, many applications, such as regression/classification models and collaborative filtering can be expressed in $\sim$30-60 lines of code in our Domain Specific Language (DSL). Even though the number of lines of code is not the most pertinent measure to compare with the convoluted design process of FPGAs, it provides a tangible intuition about the implementation complexity. Our solutions are described in in-depth detail in the following chapters.

# CHAPTER 2
# ACCELERATING MACHINE LEARNING: THE ALGORITHMIC FULL-STACK APPROACH

## 2.1 Summary

A growing number of commercial and enterprise systems increasingly rely on compute-intensive machine learning algorithms. While the demand for these compute-intensive applications is growing, the performance benefits from general-purpose platforms are diminishing. Even though hardware specialization offers a promising path, acceleration still requires long development cycles and extensive expertise in hardware design. To tackle this challenge, instead of designing an accelerator for a machine learning algorithm, we present TABLA, a framework that *generates* accelerators for a class of machine learning algorithms. The key is to identify the commonalities across a wide range of machine learning algorithms and utilize this commonality to provide a high-level abstraction for programmers. TABLA leverages the insight that many learning algorithms can be expressed as a stochastic optimization problem. Therefore, learning becomes solving an optimization problem using stochastic gradient descent that minimizes an objective function over the training data. The gradient descent solver is fixed while the objective function changes for different learning algorithms. TABLA provides a template-based framework to accelerate this class of learning algorithms. Therefore, a developer can specify the learning task by *only* expressing the gradient of the objective function using our high-level language. TABLA then automatically generates the synthesizable implementation of the accelerator for Field Programmable Gate Array (FPGA) realization using a set of hand-optimized templates. We use TABLA to generate accelerators for ten different learning tasks targeted at a Xilinx Zynq FPGA platform. We rigorously compare the benefits of FPGA acceleration to multi-core CPUs (ARM Cor-

tex A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650 Ti, and Tesla K40) using real hardware measurements. TABLA-generated accelerators provide 19.4× and 2.9× average speedup over the ARM and Xeon processors, respectively. These accelerators provide 17.57×, 20.2×, and 33.4× higher Performance-per-Watt in comparison to Tegra, GTX 650 Ti and Tesla, respectively. These benefits are achieved while the programmers write less than 50 lines of code.

## 2.2 Introduction

Both the industry and the research community are focusing on programmable accelerators, which can provide large gains in efficiency and performance by restricting the workloads they support [4, 7, 8, 9, 10, 11]. Using FPGAs as programmable accelerators has the potential for significant performance and efficiency gains while retaining some of the flexibility of general-purpose processors [40]. Commercial platforms incorporating general purpose cores with programmable logic are beginning to appear [41, 42], such as Microsoft has deployed FPGAs to accelerate their Bing search service [4]. FPGA's increasing availability and flexibility makes them an attractive platform to accelerate machine learning algorithms. However, a major challenge in using FPGAs is their programmability. Development with FPGAs still requires extensive expertise in hardware design and implementation and the overall design cycle is long even for experts [4]. This paper aims to tackle this challenge for an important class of machine learning algorithms by presenting the TABLA[1] framework. TABLA is template-based solution–from programming model to circuits–that uses FPGAs to accelerate statistical machine learning. The objective of our solution is to devise the necessary programming abstractions and an automated framework that is uniform across a range of machine learning algorithms. TABLA aims to avoid exposing software developers to the details of hardware design by leveraging commonalities in these machine learning algorithms.

[1] **T**emplate-based **A**ccelerator **B**uilder for **L**earning **A**lgorithms.

While developing TABLA, we leveraged the insight that many learning algorithms can be expressed as stochastic optimization problems [18]. Examples of such learning algorithms are support vector machines, logistic regression, least square models, backpropagation, conditional random fields, recommender systems, Kalman filters, linear and nonlinear regression models, and softmax functions. These types of learning algorithms can be optimized using stochastic gradient descent[19]. That is, the learning task becomes solving an optimization using stochastic gradient descent that iterates over the training data and minimizes an objective function. Although the objective function varies for different learning algorithms, the stochastic gradient descent solver is fixed. Therefore, the accelerator for these learning tasks can be implemented as a template design, uniform across a class of machine learning algorithms. This template design comprises the general framework for stochastic gradient descent.

TABLA automatically specializes the template design for a specific learning task by generating and integrating the hardware blocks that implement the gradient of the objective function. Therefore, a developer can specify the learning task by *only* writing the gradient of the objective function using our high-level language. The gradient function can be implemented with less than 50 lines of code for logistic regression, support vector machines, recommender systems, backpropagation, and linear regression. TABLA automatically generates a concrete accelerator (synthesizable Verilog code) for the specific learning algorithm using a set of hand-optimized template designs while considering high-level design parameters of the target FPGA. To this end, our work makes the following contributions:

(1) We observe that many common machine learning algorithms can be represented as stochastic optimization problems. This observation enables TABLA to provide a high-level, intuitive, uniform, and automated abstraction to use FPGAs to accelerate an important class of machine learning algorithms.

(2) Using this observation, we develop a comprehensive solution–from programming model to circuits–that abstracts away the details of hardware design from the programmer, yet

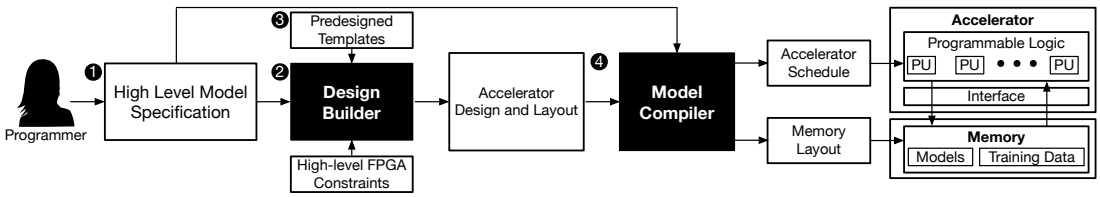



**Figure 2.1:** Overview of TABLA's workflow. The programmer only provides the gradient of the objective function, representing the learning algorithm, in TABLA's high-level programming language. The other major components of TABLA are: (a) the *design builder* that automatically generates the synthesizable Verilog implementation of the accelerator from a set of pre-designed templates; and (b) the *model compiler* that generates the execution schedule for the accelerator, the memory layout, and the memory access schedule.

generates accelerators for a range of machine learning algorithms.

(3) We use TABLA to generate accelerators for five different learning algorithms–logistic regression, SVM, recommender systems, backpropagation and linear regression–each with two different topologies. We use TABLA to generate ten different accelerators for these ten different learning tasks and evaluate them on the Xilinx Zynq FPGA platform.

We rigorously compare the benefits of the FPGA implementation to both multicore CPUs (ARM Cortex A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650 Ti, and Tesla K40), using real hardware measurements. TABLA generated accelerators provide 19.4× and 2.9× average speedup over the ARM and Xeon processors, respectively. These accelerators provide 17.57×, 20.2×, and 33.4× higher Performance-per-Watt in comparison Tegra, GTX 650, and Tesla, respectively. These results suggest that TABLA takes an effective step toward widespread use of FPGAs as an accelerator of choice for machine learning algorithms.

## 2.3 Overview

Machine learning generally involves two phases: the learning phase and the prediction phase. The learning phase, which is precursory to the prediction phase, generates a model that maps one or more inputs (independent variables) onto one or more outputs (dependent variables). The generated model is used in the prediction phase to predict the dependent

**Table 2.1:** Machine learning algorithms, their objective function, and the gradient of this objective function (used with **T**TABLA). The $\delta()$ operator in the objective and gradient functions represents a complex nonlinear transformation. For example, in logistic regression $\delta(W, X_i)$ is $sigmoid(\sum_i X_i \times W)$.

| Machine Learning Algorithm | Objective Function ($f$) | Gradient of the Objective Function ($\partial f$) |
|---|---|---|
| **Logistic Regression** | $\sum_i \{ Y_i \log(\delta(W, X_i)) + (1 - Y_i) \log(1 - \delta(W, X_i)) \} + \lambda \|\|W\|\|$ | $(\delta(W, X_i) - Y_i)X + \lambda W$ |
| **Support Vector Machines** | $\sum_i \{ 1 - Y_i W X_i \} + \lambda \|\|W\|\|$ | $Y_i X_i + \lambda W$ |
| **Recommender Systems** | $\sum_{i,j} \{ (Y_{ij} - W_j X_i)^2 \} + \lambda \|\|W, X\|\|$ | $\sum_i (Y_{ij} - W_j X_i) X_i + \lambda W$ , $\sum_j (Y_{ij} - W_j X_i) W_j + \lambda X$ |
| **Backpropagation** | $\sum_i \sum_k \{ (Y_i^k \log(\delta(W, X_i)^k)) + (1 - Y_i^k) \log(1 - \delta(W, X_i)^k) \} + \lambda \|\|W\|\|$ | $\sum_k (\alpha_i^k \Delta_i^k) + \lambda W$ |
| **Linear Regression** | $\sum_i \{ \frac{1}{2} (W X_i - Y_i)^2 \} + \lambda \|\|W\|\|$ | $(W X_i - Y_i) X_i + \lambda W$ |

variables for new unseen inputs. The learning phase is more compute intensive and can benefit significantly from acceleration. Therefore, TABLA aims to provide a comprehensive solution from programming model down to circuits that can automatically generate accelerators for the learning phase of a class of machine learning algorithms as illustrated by Figure 2.1. We briefly discuss each component of TABLA below.

❶ **High-level programming model.** TABLA provides a high-level programming model that enables programmers to specify the gradient of the objective function which defines the learning algorithm. This mathematical function captures the learning algorithm. TABLA focuses on a class of learning algorithms that can be solved using stochastic gradient descent. The stochastic gradient descent solver is uniform across a range of machine learning algorithms and therefore, the gradient function is sufficient to generate the entire accelerator design. The programmer also provides the initial and meta-parameters of the learning algorithm.

❷ **Design builder.** After the programmer provides the gradient of the objective function, TABLA's *design builder* generates the accelerator and its interfacing logic to the external memory. The design builder uses a set of pre-designed templates to generate the accelerator. The output of the design builder is a set of synthesizable Verilog designs that concretely implements the accelerator. The inputs to the design builder are (1) the gradient function, (2) a high-level specification of the target FPGA, and (3) a set of pre-designed accelerator templates in Verilog. The FPGA specification constitutes the number of DSP slices, the

number of SRAM structures (Block RAMs), the capacity of each Block RAM, the number of Block RAM read/write ports, and the off-chip communication bandwidth.

❸ **Pre-designed templates.** The design builder generates the accelerator design from a set of *pre-designed templates*. These templates are generic and uniform across a large class of statistical machine learning algorithms and support all the language constructs defined in TABLA's programming interface. The templates provide a general structure for the accelerator without making it specific to a certain algorithm or FPGA specification. The templates also contain the implementation for stochastic gradient descent, which is uniform across all the target machine learning algorithms. These predefined templates are designed by expert hardware designers and comprise both the accelerator and the interfacing logic that connects the accelerator to the rest of the system (e.g., memory).

❹ **Model compiler.** Another component of TABLA is the *model compiler* that statically generates an execution schedule for the accelerator and significantly simplifies the hardware. The inputs to the model compiler are (1) the structure of the accelerator and (2) the specification of the gradient function. The model compiler converts the gradient function to a dataflow graph and augments it with the dataflow graph of stochastic gradient descent. Then, it uses a Minimum Latency Resource-Constrained Scheduling algorithm [43] to generate the accelerator schedule. The model compiler also generates an order for the model parameters that will be learned. This order determines the layout of parameters in the memory and streamlines the interfacing logic that communicates with the memory. Finally, the model compiler generates the schedule for the memory interface.

As Figure 2.2 illustrates, TABLA uses stochastic gradient descent as the abstraction between hardware and software. This abstraction is basis for the templates from which TABLA generates the accelerator and can potentially target different platforms, including Xeon Phi, GPUs, FPGAs, CGRAs and ASICs. Specific backends need to be developed to support each of these platforms. In this paper, we focus on FPGAs since they represent a middle-ground between the efficiency of ASICs and programmability of CPUs. Next we

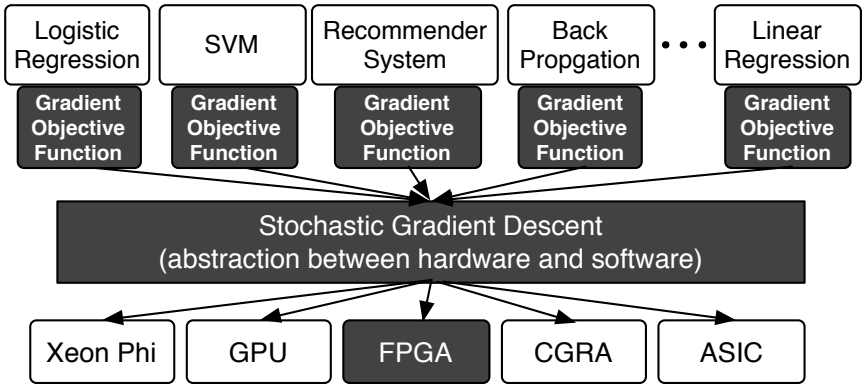



**Figure 2.2:** TABLA leverages stochastic gradient descent as an abstraction between hardware and software to create a unified framework to accelerate a class of machine learning algorithms. The highlighted blocks are the focus of this work.

**Table 2.2:** TABLA's language constructs that enable convenient representation of a wide class of learning algorithms.

| Type | Connotation | Keyword |
|---|---|---|
| **Data Declarations** | Learning model inputs | `model_input` |
| | Learning model outputs | `model_output` |
| | Learning model parameters | `model` |
| | Gradient of the objective function | `gradient` |
| | Iterator variable | `iterator` |
| **Mathematical Operations** | Basic operations | `+,-,*,/` |
| | Group operations | `pi, sum, norm` |
| | Nonlinear transformations | `gaussian, sigmoid, …, log` |

discuss each of the components of TABLA for FPGA platforms in detail.

## 2.4 Programming Through a Domain Specific Language

In TABLA, the programmer expresses machine learning algorithms by specifying the gradient of the objective function. The programmer uses our high-level programming interface[2] to specify this gradient function. Our programming interface provides the flexibility to represent a wide range of machine learning algorithms and possesses the following properties: (1) it is a high-level language that enables the representation of learning algorithms in a fashion that is familiar to machine learning experts and is close to their mathematical formulation (e.g., Table 2.1); and (2) it incorporates language constructs that are com-

[2]The details of TABLA's domain specific language, its formal syntax, grammar, and semantics of each construct are available at `http://act-lab.org/artifacts/tabla/`.

monly seen in a wide class of statistical learning algorithms. The interface comprises two language constructs: data declarations and mathematical operations. Data declarations, detailed in Section 2.4.1, allow the programmer to express different data types that represent the training data and model parameters. Further, the mathematical operations, described in Section 2.4.2, enable the programmer to declare different numerical operations used to calculate the gradient of an objective function. Table 2.2 summarizes these language constructs.

### 2.4.1 Data Type Qualifiers

Data declarations enable the programmer to specify different data types used in the gradient of the objective function. These data types include: model input, model output, model parameters, gradient, and iterators. The data declarations emphasize the different semantics held by the data in a machine learning algorithm. For example, the model_input keyword refers to an input vector (independent variables) while the model_output declaration refers to its corresponding output vector (dependent variables). Both model_input and model_output together form the training data. Both these data types are inputs to the learning algorithm while the algorithm learns the model. The gradient keyword declares the gradient of the objective function. Further, the model keyword declares the model parameters that are updated every iteration in accordance with the gradient of the objective function. Finally, the iterator keyword identifies arrays, their dimensions, and their operations. The following code snippet illustrates the use of iterators.

```
...
iterator i[0:n-1];       //iterator for arrays
Q[i] = A[i] * B[i];     //element-by-element multiplication
s = sum [i](Q[i]);      //group summation
...
```

In this example, i is an iterator variable that ranges from 0 to n-1 and can iterate over arrays with n elements starting from index 0. For example, Q[i] = A[i] * B[i] statement uses i to perform an element-by-element multiplication between the two arrays, both of size n.

Moreover, iterators can imply the autonomy of the array operations. For instance, the A[i] * B[i] cane parallelized over all the values of i. Iterators are also used in group operations to identify the array of operands. In the above example, sum[i](Q[i]) denotes that all the elements of Q need to be summed together.

As discussed, these data declarations enable programmers to specify the semantics and characteristics of different data elements in learning algorithms. Another major component of the learning algorithms is the mathematical operations that are defined over these data elements. Below, we discuss the language constructs that support these mathematical operations.

### 2.4.2 Mathematical Operations

Our language supports three types of mathematical operations: basic operations, group operations, and nonlinear transformations.

**Basic operations.** These are basic mathematical operations such as -, +, *, and /.

**Group operations.** These operations are performed over a group of elements and include sum ($\sum$), pi ($\Pi$), and norm ($||\ ||$). Besides an operand, group operations require an iterator argument. The iterator specifies the elements on which the calculation is performed. These operations generate an output with dimension one less than the input operand's dimensionality. For instance, summing the elements of a one-dimensional array generates a scalar.

**Nonlinear transformations.** These mathematical operations apply nonlinear functions (e.g., log, sigmoid, gaussian) over their operands. Since the transformation is applied to each element individually, the output has the same dimensions as the input.

Using these mathematical operations and data declarations, programmers can specify a wide range of learning algorithms at a high level without delving into the details of hardware implementation. We further demonstrate the capabilities of the language using a

concrete example that implements logistic regression.

### 2.4.3 Example: Logistic Regression

As mentioned before, the programmer only needs to specify the gradient of the objective function for the learning algorithm. Equation 2.1 shows this gradient for logistic regression in its mathematical form.

$$G_{n\times m} = \Big(\Big[\forall j \in [0,n) \Big| sigmoid(\sum_{i=0}^{m-1} X_i \times W_{j,i})\Big]_{1\times n} - Y'_{1\times n}\Big)^T_{n\times 1} \times X_{1\times m} + lambda \times W_{n\times m} \quad (2.1)$$

In this equation, $G$ is the gradient matrix with $n$ rows and $m$ columns; $X$ is an input vector with $m$ elements; $Y'$ is the expected output vector with $n$ elements; $W$ is the matrix with $n \times m$ elements that contains the model parameters; and $lambda$ is the regularization factor, which is a scalar. The following listing shows how a complex mathematical function is implemented in a textual format using TABLA's programming interface.

```
m = 53 //number of input features
n = 3  //number of model outputs
lambda = 0.1 //regularization factor

model_input  X[m]; //model input
model_output Y'[n]; //model output
model        W[n][m]; //model parameters
gradient     G[n][m]; //gradient

iterator i[0:m - 1];      //iterator for group operations
iterator j[0:n - 1];      //iterator for group operations

//m parallel multiplies followed by
//an adder tree; repeated n times in parallel
S[j] = sum [i](X[i] * W[j][i]);

Y[j] = sigmoid (S[j]); //n parallel sigmoid operations
```

```
E[j] = Y[j] - Y'[j]; //n parallel subtractions
G[j][i] = X[i] * E[j]; //n*m parallel multiplications
V[j][i] = lambda * W[i][j]; //n*m parallel multiplications
G[j][i] = G[j][i] + V[j][i]; //n*m parallel additions
```

The code above shows how this gradient function can be expressed in a few lines using TABLA's programming language. The data declarations (e.g., model_input, model_output, model, and gradient) identify the semantics of different data types. The rest of the textual representation has a close correspondence to the mathematical formulation in Equation 2.1. The gradient formula is simply broken down to multiple statements that correspond to the mathematical operations. This correspondence and the simplicity of the statements makes programming with TABLA convenient for machine learning programmers.

In the code, the two iterators i and j correspond to the subscripts in Equation 2.1 and are used to iterate over the elements of the input (X) and output (Y) vectors as well as the matrices that store the model (W) and the gradient (G). The sum statement represents the $\sum_{i=0}^{m-1} X_i \times W_{j,i}$ part of the gradient. The iterator for sum is i, similar to the formula in which $\sum$ iterates over $i$. This statement first performs the multiplication $X_i \times W_{j,i}$ and then accumulates all the multiplication results into a single result S[j] assuming a constant j. The left hand side of the statement, S[j], mandates that the accumulation needs to be repeated n times using the j iterator. Next, the sigmoid statement continues the gradient calculation as shown by $\left[\forall j \in [0,n) \middle| sigmoid(\sum_{i=0}^{m-1} X_i \times W_{j,i})\right]_{1\times n}$. The rest of the code similarly performs the remaining part of the gradient computation. At the end, the G[j][i] variable in the code corresponds to the elements of the $G_{n\times m}$ matrix in Equation 2.1.

A wide range of machine learning algorithms can be represented using TABLA's programming interface. Furthermore, the programming interface can be easily extended to accommodate the representation of an even wider range of learning algorithms. Although MATLAB and R can also be used to represent the same learning algorithms, we designed and used TABLA's own programming interface because of: (1) easier representation of gradient functions using the common mathematical constructs used in machine learning; (2) clear-cut identification of parallelism in the code; and (3) convenient conversion of gradi-

ent function into the final hardware design using our model compiler, described in the next section. We are also working on providing translators that convert MATLAB and R code to TABLA's language.

## 2.5 Compilation Workflow of TABLA

TABLA's model compiler statically generates an execution schedule for the accelerator using the gradient of the objective function provided by the programmer. The model compiler accomplishes this task in three steps. (1) The first step integrates the gradient of the objective function with the stochastic gradient descent solver. (2) In the second step, the compiler generates an intermediate representation, i.e., the Dataflow Graph (DFG) of the entire learning algorithm. (3) Finally, in the last step, the compiler translates this Dataflow Graph (DFG) into a static schedule for hardware execution. We specifically use static scheduling since it simplifies the hardware and improves the efficiency of the accelerated execution. Each of these compilation steps are described in further detail in this section.

### 2.5.1 Integrating Stochastic Gradient Descent

After the programmer provides the gradient of the objective function, TABLA uses the stochastic gradient algorithm to learn the model parameters from the training data. Learning a model from the training data requires a solver that finds the minimum value of the objective function that represents the learning algorithm. Since stochastic gradient descent is independent of the learning model, we devise a general template to implement it. TABLA integrates this template with the programmer-provided gradient code using the gradient and model variables. These keywords explicitly identify the inputs to the stochastic gradient descent solver. The following code snippet shows the template code of the stochastic gradient descent solver.

```
gradient    G[n][m]; //gradient
model       W[n][m]; //model parameters
```

```
iterator i[0:m - 1]; //iterator for group operations
iterator j[0:n - 1]; //iterator for group operations

G[j][i] = u * G[j][i] //n*m parallel multiplications
W[j][i] = W[j][i]-G[j][i]; //n*m parallel subtractions
```

As the code shows, the model parameters (W[n][m]) are updated in the opposite direction of the gradient (G[n][m]) with a rate u, called the learning rate. Once the gradient function is integrated with the stochastic gradient descent solver, the model compiler generates the DFG of the entire learning algorithm.

### 2.5.2 Generating Dataflow Graph

TABLA's model compiler converts any code written in our programming interface into a dataflow graph. Each language construct corresponds to a small and simple DFG. The model compiler scans the code and replaces each construct with its corresponding DFG. The compiler then links these small DFGs to create the final DFG for the learning algorithm.

**Dataflow graph of individual operations.** Figure 2.3 shows sample DFGs of three types of mathematical operations that are supported by TABLA's language: basic, group, and nonlinear. The nodes are basic computations and the edges capture the dependencies. The group operations require more than one computational node. As Figure 2.3 depicts, the DFG for sum is an adder tree and the DFG for norm includes a layer of multiplications that feed into an adder tree. The DFG also captures the parallelism amongst the basic computations and enables the model compiler to generate an efficient execution schedule for the accelerator.

**Dataflow graph of the learning algorithm.** Figure 2.4 shows the complete DFG for logistic regression. This DFG corresponds to the example code provided in Section 2.4.3 when n is 1. As the figure illustrates, the model compiler combines and links the DFG

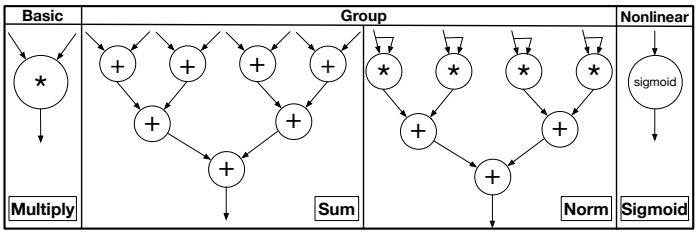



**Figure 2.3:** Dataflow graph of basic, group, and nonlinear operations.

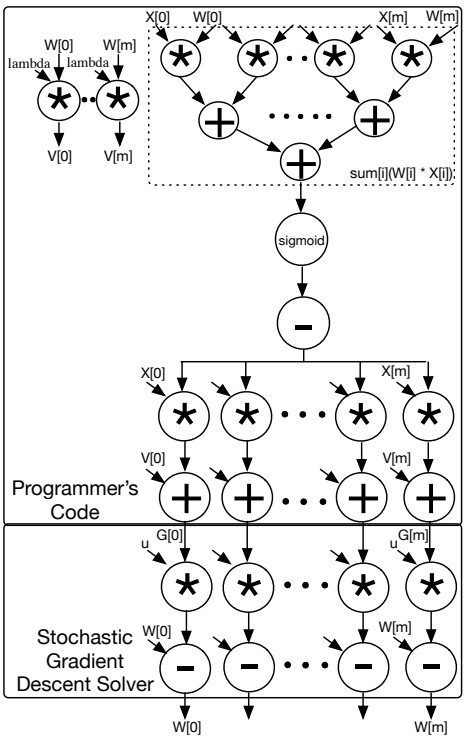



**Figure 2.4:** Complete dataflow graph of the logistic regression algorithm.

of each operation to generate the entire DFG. For example, the compiler converts the sum[i](X[i]*W[j][i]) statement to a series of multiplications followed by an adder tree, which is the DFG for sum (Figure 2.3). Translating code to DFG is straightforward since the dataflow graphs of each operation is predetermined. As shown at the bottom of Figure 2.4, the DFG of logistic regression also includes the computation of stochastic gradient descent solver.

### 2.5.3 Static Scheduling

Once the DFG is generated, the model compiler statically generates a step-by-step schedule of each operation. We use the Minimum Latency–Resource Constrained Scheduling (ML–RCS) algorithm [43] to generate this schedule[3]. This algorithm aims to optimize for minimum execution latency while being constrained by the limited set of resources available on the accelerator platform. Algorithm 1 presents this scheduling algorithm.

**Inputs:** $R$: Set of available resources
$O$: Set of all the operations to be scheduled
$D$: Distance to sink for each operation
**Output:** $S$: Final schedule
Initialize $S \leftarrow \emptyset$
Initialize $current\_cycle \leftarrow 0$
**while** ($O \neq \emptyset$) **do**
  **for** ($r \in R$) **do**
    **if** $o \in O$ $where$ $o$.predecessors = DONE & $o$.distance = max($D$) **then**
      $schedule$.op = $o$; $schedule$.resource = $r$; $schedule$.cycle = $current\_cycle$
      $S$.append($s$)
      $O$.remove($o$)
    **end if**
  **end for**
  $current\_cycle$ = $current\_cycle$ + 1
**end while**

**Algorithm 1: Minimum-latency resource constrained scheduling.**

To understand the algorithm, we first define a property of each operation ($o$) called *distance from sink*, denoted as "$o$.distance" in the Algorithm 1. *Distance from sink* of $o$ is the number of dependent operations between $o$ and the final output or the sink of the DFG. This distance captures the criticality of an operation. The higher an operation's *distance from sink*, higher its scheduling priority. Algorithm 1 picks an available resource $r$ and schedules an operation $o$ at the $current\_cycle$ on $r$ if the following two conditions are satisfied: (1) all the predecessors of $o$ have already been scheduled, i.e., $o$ is ready; and (2) $o$ is on the critical path of execution, i.e., its *distance from sink* is the maximum

[3] Note that the DFG itself represents the As-Soon-As-Possible (ASAP) schedule of the operations. In the ASAP schedule, an operation is scheduled for computation as soon as it is ready, i.e., all of its predecessors have finished their computation. The ASAP schedule provides minimum latency; however, it assumes infinite resources. We use ML–RCS since it considers the limited availability of compute resources.

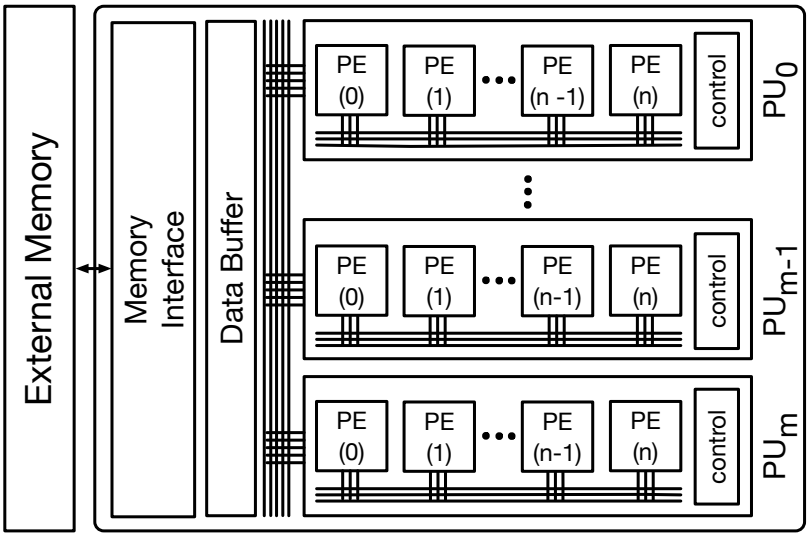



**Figure 2.5:** Template design for the accelerators; it is a scalable, general, modular, and highly customizable architecture. The design builder shrinks or expands the template architecture based on the requirements of the DFG and the availability of resources on the target FPGA. This hierarchical design is clustered into a set of PUs that comprise of a number of PEs. The PU are connected through an inter-PU bus that is also connected to the memory interface. The PEs use a dedicated intra-PU bus to communicate.

among all unscheduled ready operations. The algorithm picks the next available resource or increments the $current\_cycle$ if all the resources are being utilized. The algorithm terminates when all the operations are scheduled. To generate this schedule, TABLA's design builder first needs to generate the skeleton of the accelerator and determine the number of available resources. The next section discusses this process and the template architecture of the accelerator.

## 2.6 TABLA's Design Builder and Template-Based Designs

### 2.6.1 Design Builder

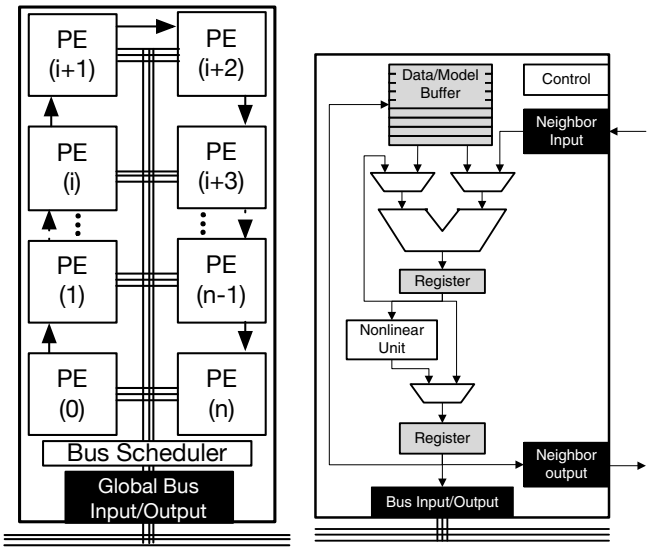
TABLA's design builder generates synthesizable Verilog code of the learning accelerator given the DFG and schedule of the learning algorithm, and the high-level specification of the target FPGA. The FPGA specification comprises the number of DSP slices, the ALU operations supported in the DSP slices, the number of SRAM structures (Block RAMs),

the capacity of each Block RAM, the number of read/write ports on a Block RAM, and the off-chip communication bandwidth. Given this information and the DFG of the learning algorithm, the design builder customizes our hand-optimized template accelerator architecture for the specified machine learning algorithm.

As Figure 2.5 shows, the template design is a clustered hierarchical architecture. A series of Processing Units (PUs) that include a set of Processing Engines (PEs) constitute this hierarchical architecture. This clustered template architecture is scalable, general, and highly customizable. The design builder shrinks or expands this template design considering the degree of parallelism in the DFG and the availability of the resources in the target FPGA. The design builder first extracts the maximum number of parallel operations from the DFG and select the total number of the PEs accordingly. Based on the DFG, the design builder also determines the ALU operations and the nonlinear transformation units that need to be included in the PEs. If an ALU operation or a nonlinear function is not used in the DFG, the corresponding hardware unit is excluded from the final accelerator design. The design builder also generates the control unit of the PEs, PUs, and the buses according to the schedule of operations. The scheduling algorithm and the design builder work in tandem. The design builder determines the number of PEs (compute resources) for the scheduler to generate the execution schedule. Then, based on the schedule, the design builder generates the control logic. The design builder also determines the number of PEs per each PU depending on the target FPGA as we will discuss in Section 2.7.2.4. Finally, the design builder adds the memory interface unit and generates the access schedule to the memory according to the execution schedule. The remainder of this section discusses the PU and the PE designs.

### 2.6.2 Template Design for Processing Units

As shown in Figure 2.5, the processing units construct the first level of hierarchy in our template design. The PUs are self-contained structures that comprise a set of identical pro-




**(a)** Processing Unit (PU) **(b)** Processing Engine (PE)

**Figure 2.6:** (a) Template PU design comprising a set of PEs that are connected through an intra-PU bus. This bus is also connected to the global inter-PU bus. (b) Template PE design with ALU, control logic, data buffer, nonlinear unit, and the links to the neighboring PEs.

cessing engines as depicted in Figure 2.6a. Grouping PEs as PUs makes the template design modular and localizes the majority of data traffic within PUs. Both modularity and locality of traffic enhances the scalability and the customizability of the template design. This characteristic of the template enables the design builder to generate a concrete accelerator design with any number of PUs. Conceptually, a single PU can carry out the computation of an entire learning algorithm. However, the design builder scales up the number of PUs if the DFG can utilize the additional resources. The PUs are connected through a pipelined global bus. The communication between PUs is statically orchestrated by the model compiler and is loaded into the PUs as part of the accelerator configuration. The PUs are also connected to the memory interface through a data buffer. The PUs are merely consumers of data and do not initiate requests. The data buffer fetches data from the external memory and sends the data to the PUs according to a static schedule generated by the model compiler. The static scheduling of communication significantly simplifies the PU design and the busing logic. This hardware-software co-design approach makes the accelerator template design scalable and enables the design builder to cater the needs of a variety of

learning algorithms.

### 2.6.3 Template Design for Processing Engine

Figure 2.6b depicts the template design of the processing engines. PEs are the basic blocks of our template design and are customized according to the DFG of the learning algorithm. As illustrated, each PE contains an ALU that performs the calculations and a local memory (data/model Buffer) that stores the model parameters and data elements. Some of the components are fixed within a PE, while the others are customizable based on the learning algorithm's DFG.

The *fixed components* in a PE are the ALU, data/model buffer, registers, and busing logic. All the learning algorithms have some form of mathematical operations, making the ALU a fixed component. Although the ALU is fixed, the operations that it supports changes according to the learning algorithm's DFG. Additionally, a buffer is necessary to store the model parameters or any other incoming data. The buffer retains the model parameters that are updated (learned) during the execution. The PE's share of training data is also stored in this buffer. The registers are essential to a PE as they enable the storage of intermediate results. Finally, bus interfaces are always needed to channel the incoming data from memory or other PEs and PUs.

The highly *customizable components* in the PE are the control unit, the nonlinear unit, and the neighbor input/output communication links. Firstly, the control unit stores the PE's schedule of operations. This schedule is a queue of predetermined control signals that directs different components of the PE. This schedule changes with the DFG of the learning task. Secondly, the nonlinear unit is not required by some algorithms such as SVM, recommender system, and linear regression. This unit is excluded or customized according to the algorithm. Finally, communication between neighboring PEs is only useful for learning algorithms that aggregate data (e.g., use sum ($\sum$) or pi ($\Pi$)). During aggregation, the short direct links between neighboring PEs enable parallel exchange of data without serializing

**Table 2.3:** Benchmarks, their brief description, size of the training datasets, number of input features, model topology, lines of code to express the gradient function of the learning algorithm with TABLA's programming interface, and the number of PEs in the TABLA generated accelerators.

| Name | Model | Algorithm Name | Description | Input Vectors | # of Features | Model Topology | Lines of Code | # of PEs |
|---|---|---|---|---|---|---|---|---|
| LogisticR | M1 | Logistic Regression | Estimates the probability of dependent variable given one or more independent variables | 581,000 | 54 | 54 | 20 | 32 |
| | M2 | | | 500,000 | 200 | 200 | 20 | 64 |
| SVM | M1 | Support Vector Machines | Classifies data into different categories by identifying support vectors | 581,000 | 54 | 54 | 23 | 32 |
| | M2 | | | 500,000 | 200 | 200 | 23 | 64 |
| Reco | M1 | Recommender Systems | Information filtering system that predicts the preference a user would give to an item | 1,700,000 | 2,700 | 27,000 | 31 | 32 |
| | M2 | | | 24,000,000 | 10,000 | 100,000 | 31 | 64 |
| Backprop | M1 | Backpropogation | Trains a neural network that models the mapping between the inputs and outputs of the data | 38,000 | 10 | 10 -> 9 -> 1 | 48 | 64 |
| | M2 | | | 90,000 | 256 | 256 -> 128 -> 256 | 48 | 64 |
| LinearR | M1 | Linear Regression | Models relationship between a dependent variable and one or more explanatory variables | 10,000 | 55 | 55 | 17 | 64 |
| | M2 | | | 10,000 | 784 | 784 | 17 | 64 |

computation by requiring PEs to contend for the intra-PU bus. Once the design builder customizes the PE design in congruence with the DFG of the learning algorithm, it groups the PEs as PUs and generates the final concrete accelerator as synthesizable Verilog code.

In the next section, we evaluate TABLA-generated accelerators for ten different learning tasks.

## 2.7 Evaluation

We evaluate TABLA using an off-the-shelf Xilinx Zynq ZC702 FPGA platform, specifications of which are summarized in Table 2.5. We synthesize the TABLA-generated accelerators with 64-bit Vivado v2015.1. The accelerators are connected to the external memory via four Xilinx Advanced eXtensible Interface (AXI) controllers and operate at 100 MHz. We compare the performance and energy benefits of these FPGA accelerators to a diverse set of high-performance and low-power CPUs and GPUs. We use hardware measurements to rigorously compare the accelerator benefits to both multicore CPUs (ARM Cortex A15 and Xeon E3) and many-core GPUs (Tegra, GTX 650 Ti, and Tesla K40).

### 2.7.1 Experimental Setup

#### *2.7.1.1 Benchmarks and Training Datasets*

Table 2.3 lists the machine learning algorithms used to evaluate TABLA. We study five popular machine learning algorithms: Logistic Regression (LogisticR), Support Vector Machines (SVM), Recommender Systems (Reco), Backpropagation (BackProp), and Linear Regression (LinearR). These algorithms represent a wide range of learning algorithms encompassing regression analysis, statistical classification, information filtering systems, recommender systems, and artificial neural networks. Table 2.3 also includes some of the most pertinent learning parameters such as the number of training vectors, the model topology, the number of lines required to implement the gradient function in the TABLA programming interface, and the number of PEs/PUs constituting the design of each algorithm. Each algorithm is evaluated with two model topologies M1 and M2. The evaluation across multiple models allows us to evaluate the flexibility of the TABLA framework to accommodate changes in the topology of a machine learning algorithm. For LogisticR and SVM, we use two model topologies from the UCI repository[44]. One dataset comprises 54 features and the other dataset consists of 200 features. We modified the datasets to incorporate binary output values. For Reco, we use two different topologies from movieLens [45, 46], a movie database. For BackProp we use two topologies: a large neural network topology (256→128→256) [47] and a small neural network topology (10→9→1) [48]. For LinearR we use one topology from the UCI repository and one from MNIST [49]. The Input Vectors column in Table 2.3 shows the number of input vectors in each training data set. The # of Features column denotes the number of independent variables. The Model Topology column shows the topology and the number of parameters that are trained via the learning task. Finally, the Lines of Code column lists the number of lines of code that were required to implement each learning task's gradient function using TABLA's programming interface. The number of lines vary from 17 for LinearR to 48 for Backprop depending on the complexity

**Table 2.4:** Specifications of the CPU's and GPU's used to evaluate TABLA.

| Platform | Cores | Clock (MHz) | Memory (GB) | TDP (W) | Process (nm) | MSRP (USD) |
|---|---|---|---|---|---|---|
| **ARM Cortex A15** | 4+1 | 2300 | 2 | 5 | 28 | $191 |
| **Intel Xeon E3-1246 v3** | 4 | 3600 | 16 | 84 | 22 | $290 |
| **Tegra K1 GPU** | 192 | 852 | 2 | 5 | 28 | $191 |
| **NVIDIA GTX650Ti** | 768 | 928 | 1 | 110 | 28 | $150 |
| **Tesla K40** | 2880 | 875 | 12 | 235 | 28 | $5,499 |

of the gradient function for a given algorithm . These numbers suggest that TABLA establishes an effective high level programming interface that abstracts away the intricate details of hardware design from its users. Finally, the # of PEs column gives the total number of PEs for each benchmark in the final accelerator design generated by TABLA.

### *2.7.1.2 CPU and GPU Platforms*

As shown in Table 2.4, we compare TABLA-generated accelerators with two multicore CPU processors: (1) the low-power quad-core ARM A15 available on the Nvidia Jetson TK1 platform [50] that operates at 2.3 GHz; and (2) the high performance quad-core Intel Xeon E3 with hyper-threading support that operates at 3.6 GHz. We also compare TABLA-generated accelerators to three GPU processors: (1) the low-power Tegra K1 GPU, which is available on the Jetson TK1 board with 192 SIMD cores; (2) the desktop-class GeForce GTX 650 Ti with 768 SIMD cores; and (3) the high-performance Tesla K40 GPU accelerator with 2880 SIMD cores. All the platforms run Ubuntu Linux version 14.04.

**Multithreaded vectorized CPU execution.** To compare TABLA with the CPU platforms, we use optimized open-source multithreaded implementations. We use Liblinear [51] for logistic regression and SVM; MLPACK [52] for recommender systems and linear regression; and Caffe [20] for backpropagation. The code is compiled using gcc 4.8 with *-O3 -ftree-vectorize -march=native* flags in order to enable aggressive compiler optimizations and utilize vector instructions. All benchmarks use four threads on ARM and eight threads

**Table 2.5:** FPGA platform specifications.

| FPGA hardware platform | |
|---|---|
| **Model** | Xilinx Zynq ZC702 (Artix-7) |
| **Technology** | TSMC 28nm |
| **FPGA Capacity** | 53K LUTs |
| | 106K Flip-Flops |
| **Peak Frequency** | 250MHz |
| **BRAM** | 630 KB |
| **DSP Slices** | 220 Count of type DSP48E1 |
| **MSRP** | $129 |

on Xeon. The ARM CPU does not support simultaneous multithreading (SMT) while the Xeon CPU does. Multithreading support is either implemented using OpenMP (Liblinear) or using OpenBLAS [53] (MLPACK and Caffe). In addition to libraries reported in this paper, we tried a wide spectrum of other available libraries (LibFM [54], Libsvm [55], FANN [56]). However, these libraries provided inferior performance in comparison to the ones presented.

**Optimized CUDA implementation for GPU execution.** For the GPU platforms, we use highly optimized CUDA implementations from [57], Caffe+cuDNN [20], and LibSVM-GPU [58]. Caffe was configured to use the latest version of Nvidia cuDNN library [59]. The cuDNN library is a dynamic library provided as a binary without source code and is pre-optimized by Nvidia for our target GPUs. For the other benchmarks, we made our best effort to hand-tune their CUDA code for each GPU platform and optimized the number of blocks and number of threads-per-block. Moreover, all of the benchmarks are compiled separately for each GPU using target-specific flags.

**Execution time measurements.** The execution time for both CPU and GPU implementations are obtained by measuring the wall clock time, averaged over 100 runs. The CPU and GPU execution times are compared with the FPGA runtime obtained from the hardware counters synthesized on the programmable logic.

#### *2.7.1.3 Power Measurements*

We employ a variety of strategies in order to measure each benchmark's power consumption on different platforms.

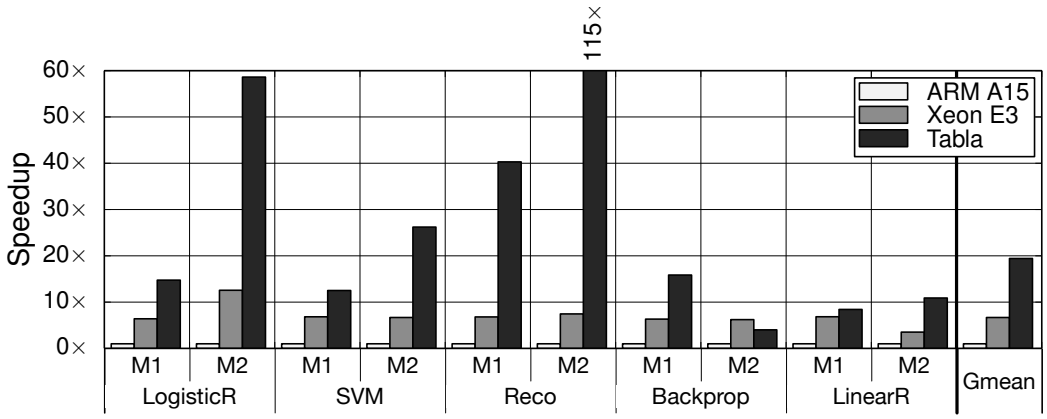
**Power measurements using vendor libraries.** For Xeon E3, we utilize the Intel Running Average Power Limit (RAPL) energy consumption counters available in the Linux kernel. For Tesla K40, we use the Nvidia Management Library (NVML) to obtain the average power while running each benchmark. GTX 650 Ti does not support the NVML library; however, GTX 650 Ti and Tesla K40 share the same microarchitecture. Hence, we make a conservative estimation of the GTX650 Ti power consumption by scaling the Tesla K40 measurements using the ratio of the two chips' Thermal Design Powers (TDPs). For each benchmark, we calculate the ratio between the measured power in Tesla and its TDP. We multiply this ratio with the GTX 650 Ti's TDP, and use 95% of the resulting value as its estimated power.

**Power measurements in hardware.** ARM Cortex A15 CPU and Tegra K1 GPU are a part of Jetson TK1's development board. Jetson TK1 does not provide a software mechanism to measure energy consumption. Therefore, we use the Keysight E3649A Programmable DC Power Supply to measure its power consumption. During each benchmark execution, we subtract the idle power from the obtained readings. The ZYNQ platform uses Texas Instruments UCD9240 power supply controllers that enable us to measure the power consumption from the power plane of the board.

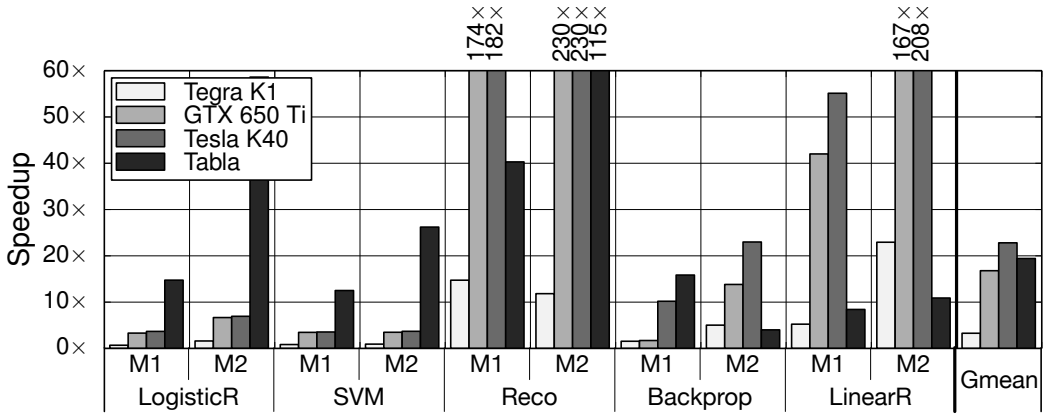
### 2.7.2 Experimental Results

#### *2.7.2.1 Performance Comparison*

**Comparison with CPUs.** Figure 2.7a shows the speedup of TABLA-generated FPGA accelerators and the Xeon E3 CPU when compared with the ARM A15 CPU. ARM is the baseline in all the speedup graphs. Henceforth, we refer to the TABLA-generated accelerators as TABLA. On average, TABLA outperforms ARM by 19.4× and Xeon by 2.9×. Furthermore, the high-performance Xeon is 6.7× faster than the low-power ARM. TABLA outperforms both the CPUs since our careful compiler-architecture co-design alleviates the Von Neumann overhead of instruction fetch, decode, etc. By leveraging static scheduling for both computation and memory accesses, the accelerators can carry out the calculations efficiently. In comparison to ARM, the performance improvements for TABLA range from 4× to 115×. This variation in performance benefits comes from the disparity in the model topology, which in turn leads to different levels of parallelism in the DFG. For instance, the relatively large model topology of Reco M2 provides greater opportunities for parallelism that can be exploited by the accelerator and provides the maximum speedup of 115×. On the other hand, for the Backprop M2 benchmark, TABLA provides the least speedup of 4× in comparison to ARM and a slowdown of 56% in comparison to Xeon. However, TABLA is still faster than Xeon by 2.5× for Backprop M1 benchmark. In the backpropagation algorithm, there are several dependent operations that lead to serialization of the computation and limit the opportunities for parallelism. These dependencies are not as limiting in the smaller model Backprop M1; however, their effect exacerbates as the model topology grows (Backprop M2). To overcome this challenge, one possible solution would be to simultaneously run more iterations of the gradient function over different training input vectors. Similar optimizations can be integrated into TABLA's framework owing to its cross-layer nature. Ultimately, such optimizations would be applied during the compilation stage in accordance to the DFG of the learning task in order to exploit more resources that are




**(a)** Speedup of Xeon and TABLA in comparison to ARM A15.

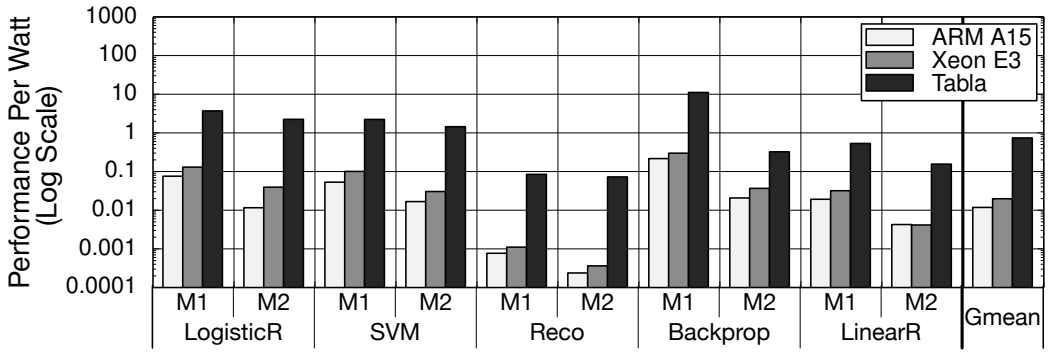



**(b)** Speedup of GPUs and TABLA design in comparison ARM A15.

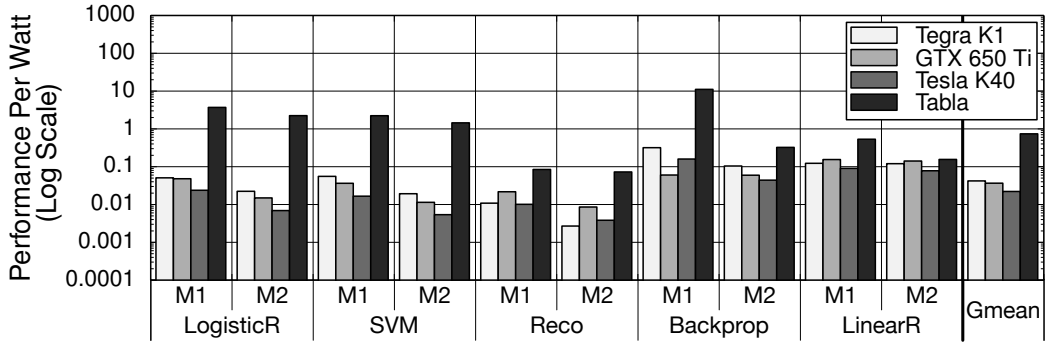
**Figure 2.7:** Speedup of **TABLA** in comparison to a diverse set of CPU and GPU platforms. The baseline is ARM A15.

available on the target FPGA platform.

**Comparison with GPUs.** Figure 2.7b depicts the speedups with different GPU platforms and TABLA. As mentioned before, ARM is the baseline. As the results show, Tesla provides an average speedup of 22.8×, followed by GTX 650 Ti with an average speedup of 16.8×. Finally, the low power Tegra K1 GPU only provides a speedup of 3.3× over ARM. In comparison to Xeon, Tesla and GTX 650 Ti provide an average speedup of 3.42× and 2.51×, respectively. However, Tegra 2.04× is slower than Xeon. These results can be attributed to the fact that Tesla (TDP of 235W), GTX 650 Ti (TDP of 110W), and Tegra (TDP of 5W) are GPUs with decreasing order of TDP (power envelope). The higher power con-




**(a)** Performance-per-Watt of ARM, Xeon and TABLA.




**(b)** Performance-per-Watt of Tegra, GTX 650 Ti, Tesla and TABLA

**Figure 2.8:** Comparison of Performance-per-Watt between CPUs, GPUs and **T**ABLA.

sumption of Tesla and GTX 650 Ti justify their higher speedup numbers. On the other hand, even though TABLA operates in a lower power budget (TDP of 2W), it marginally outperforms GTX 650 Ti by 16%. However, TABLA is surpassed by Tesla with a margin of 18%. TABLA is $5.9\times$ faster in comparison to Tegra. These results show that TABLA either follows or outperforms the GPU platforms due to its specialized hardware design tailored for a particular learning task while operating under a low power budget of 2W. For benchmarks LogisticR M1, LogisticR M2, SVM M1, SVM M2, and Backprop M1, TABLA shows higher performance than Tesla. These benchmarks have relatively small topologies and hence the coarse-grained parallelism that can be exploited by the GPUs is fairly limited. On the other hand, the TABLA-generated accelerators are able to take advantage of the available fine-grained parallelism. As the size of the topology increases, GPUs tend to exhibit higher speedups due to the availability of more computation that can be parallelize. However, GPUs that outperform TABLA require significantly higher power.

#### 2.7.2.2 *Performance-per-Watt Comparison*

The performance benefits vary significantly across the platforms as these platforms occupy different points in the performance-power design space. To understand the performance benefits for fixed energy efficiency, we use the Performance-per-Watt as a unifying metric to compare these platforms.

**Comparison with CPUs.** Figure 2.8a compares the Performance-per-Watt for ARM A15, Xeon E3 and TABLA. On average, TABLA achieves 62.7× and 37.4× higher Performance-per-Watt over ARM and Xeon, respectively. Xeon provides 67% higher Performance-per-Watt than ARM. Even though ARM is a low power CPU, Xeon shows better Performance-per-Watt due to its significantly higher performance.

**Comparison with GPUs.** Figure 2.8b illustrates the Performance-per-Watt for the GPU platforms. TABLA provides 17.57×, 20.2× and 33.4× higher Performance-per-Watt in comparison to Tegra, GTX 650 Ti, and Tesla, respectively. In comparison to Tesla, Xeon achieves just 11% higher Performance-per-Watt, however, Tesla provides much higher performance gains. Similarly, Tegra, GTX 650 Ti, and Tesla provide 3.57×, 3.1×, and 1.88× higher Performance-per-Watt than ARM, respectively, while achieving higher speedup gains. The TABLA-generated FPGA accelerators close this performance gap to a large extent and provide much higher efficiency and operate with a lower power budget. In any case, GPUs can be explored as an alternative back-end for TABLA.

As the results show, TABLA framework provides significant speedup over the multicore CPUs and higher efficiency over the many-core GPUs. These results can be attributed to the fact that TABLA streamlines the execution by generating a static schedule even for memory accesses. TABLA's compiler also tries to maximize the data transfer bandwidth by marshaling the data. It carefully lays out the parameters in local memory and data in external memory in order to reduce the accesses to the external memory.

**Table 2.6:** Resource utilization on the FPGA for each benchmark.

| Benchmark | | LUT (Total Available: 53200) | | Block RAM (Total Available: 630KB) | | Flip-Flops (Total Available: 106400) | | DSP Slices (Total Available: 220) | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Model | Total Used | Utilization | Total Used (B) | Utilization | Total Used | Utilization | Total Used | Utilization |
| LogisticR | M1 | 1873 | 3.52% | 440 | 0.07% | 1230 | 1.16% | 32 | 14.55% |
| | M2 | 3843 | 7.22% | 1612 | 0.25% | 2446 | 2.30% | 64 | 29.09% |
| SVM | M1 | 1326 | 2.49% | 440 | 0.07% | 1206 | 1.13% | 32 | 14.55% |
| | M2 | 3296 | 6.20% | 1612 | 0.25% | 2422 | 2.28% | 64 | 29.09% |
| Reco | M1 | 1326 | 2.49% | 115504 | 17.90% | 1206 | 1.13% | 32 | 14.55% |
| | M2 | 3296 | 6.20% | 439652 | 68.15% | 2422 | 2.28% | 64 | 29.09% |
| Backprop | M1 | 1916 | 3.60% | 400 | 0.06% | 648 | 0.61% | 16 | 7.27% |
| | M2 | 7672 | 14.42% | 262148 | 40.64% | 2602 | 2.45% | 64 | 29.09% |
| LinearR | M1 | 3296 | 6.20% | 444 | 0.07% | 2422 | 2.28% | 64 | 29.09% |
| | M2 | 3296 | 6.20% | 6284 | 0.97% | 2422 | 2.28% | 64 | 29.09% |

#### *2.7.2.3 Area and FPGA Utilization*

Table 2.6 shows the resource utilization for different components on the FPGA for each learning task. Backprop M1 utilizes the least area among all the learning algorithms as it has a relatively small model and requires only 16 PEs for its default configuration. On the other hand, Reco M1, Reco M2, and Backprop M2 utilize a larger area in their default configuration. These learning tasks also occupy more BRAM (FPGA Memory Slices) to accommodate the large number of parameters that need to be stored in the accelerator.

#### *2.7.2.4 Design Space Exploration*

**Number of PEs per PU.** During the development of the template-based designs, we perform a design space exploration to find the PE and PU configuration that provides the highest frequency while maintaining parallelism within each PU. Empirically, a PU design with eight PEs strikes a balance between frequency and intra-PU parallelism. Note that this design space exploration is not the responsibility of the programmer but part of TABLA.

**Number of processing engines.** While the number of PEs in each PU is fixed for the target FPGA, TABLA's design builder determines the number PUs (total number of PEs) in accordance with the algorithm's DFG. We perform a design space exploration by varying the total number of PEs. When the number of PEs exceeds eight, they are grouped

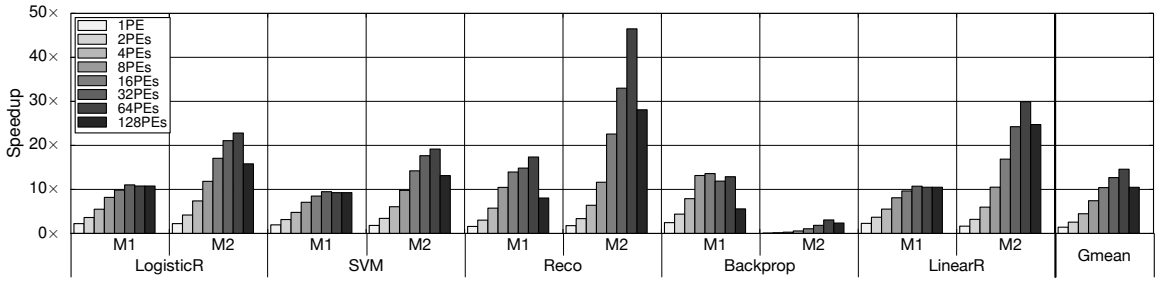



**Figure 2.9:** Speedup change for varying number of PEs in the design with ARM CPU as the baseline

into a PU with eight PEs each. Figure 2.9 shows the effect of this sweep on the speedup results. The baseline is the A15 ARM multicore CPU. As expected, the initial increase in the number of PEs leads to a linear increase in speedup. However, beyond a certain number of PEs we either observe diminishing returns or a decrease in speedup. Since the available parallelism in the algorithms is limited, increasing the number of PEs beyond a point leads to underutilization of the added PEs. For instance, for LogisticR M1, a maximum of 54 operations can be performed in parallel. Therefore, providing more than 54 PEs is inconsequential. In some cases such as LogisticR M1, increasing the number of PEs beyond 32 leads to a decrease in the speedup. When the number of PEs is greater than 32, the operational frequency decreases due to the requirement of a wider global bus. Therefore, in this case (LogisticR M1), adding more PEs does not improve performance due to the lack parallelism but rather decreases the speedup due to slower hardware. The last column in Table 2.3 shows the total number of PEs for each benchmark that are grouped in PUs with 8 PEs each.

**Bandwidth sensitivity.** Machine learning algorithms are both compute and data intensive tasks. We design the accelerators to exploit the fine-grained parallelism in the computational component of the algorithm. In addition to the compute units, the training data is streamed to the PEs from the external memory while the PEs store the model parameters locally. The AXI interfaces offer a fixed bandwidth for the training data transfer. We perform a speedup sensitivity analysis while varying the bandwidth between external memory

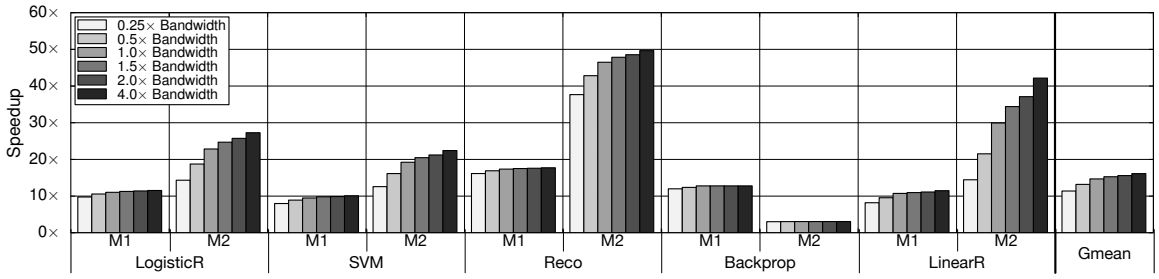



**Figure 2.10:** Speedup with varying Bandwidth for **T**ABLA generated accelerator with ARM as the baseline

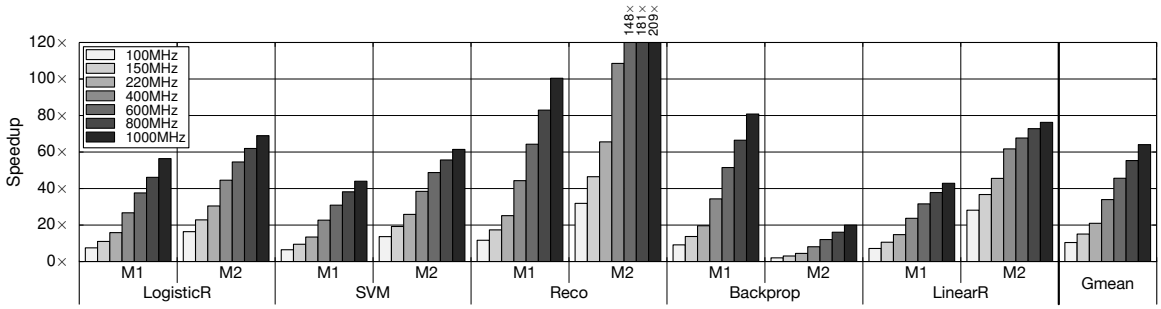



**Figure 2.11:** Speedup with varying Frequency for **T**ABLA generated accelerator with ARM as the baseline

and the accelerator. We perform the bandwidth sweeps using a cycle-accurate simulator, which is validated against the hardware. Figure 2.10 shows the speedup for each benchmark when the bandwidth varies from $0.25\times$ to $4\times$ of the default bandwidth. The baseline is ARM. The bandwidth can be a bottleneck at low values such as $0.25\times$ of the default bandwidth. As the bandwidth increases, the speedup starts to increase but we observe diminishing returns after a certain point since computation dominates the execution time. By providing a bandwidth that is $4\times$ the default value, the performance only improves by 60%. This limited improvement is in part due to the fact that the model compiler stores the most frequently accessed data (the model parameters and intermediate results) in the PE's local buffers. This limits the accesses to the external memory and attenuates the effects of external memory bandwidth on performance.

**Frequency sensitivity.** Figure 2.11 shows the speedup trends for changing FPGA frequency assuming default number of PEs and bandwidth. These results are merely a sen-

sitivity study that uses our measurements and estimate the befits assuming frequency increases in future FPGA platforms or when TABLA is used to generate ASIC designs that can operate in higher frequencies than FPGAs. Due to the high computation involved in the machine learning algorithms, increasing the frequency gives higher speedup. The speedup scales linearly with the frequency until the bandwidth becomes a bottleneck for the system.

## 2.8 Related Work

There have been several proposed architectures that accelerate machine learning algorithms [57, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 48, 72]. However, TABLA fundamentally differs from these works, as it is not an accelerator. TABLA framework generates accelerators for an important class of machine learning algorithms, which can be expressed as stochastic optimization problems. TABLA uses the commonalities across a wide range of learning algorithms and provides a high-level abstraction for programmers to utilize FPGAs as the accelerator of choice for machine learning algorithms without exposing the details of hardware design.

TABLA also automatically incorporates stochastic gradient descent solver into its learning accelerators. There have been past proposals that focus solely on accelerating gradient descent [73] and conjugate gradient descent [74, 75, 76, 73] solvers. The most recent work [73] focuses merely on designing hardware units for different linear algebra operations that are used in gradient descent and conjugate gradient solvers. However, these works do not specialize their architectures for machine learning algorithms or any specific objective function. Moreover, they neither provide domain-specific programming models nor generate accelerators.

**Machine learning accelerators.** There have been several successful works that focus on accelerating a single or a range of fixed learning tasks. Several efforts have focused on designing accelerators for a specific algorithm (K-Nearest Neighbor) [77, 61, 60]. Others

propose accelerator designs for k-Means [62, 63, 64], support vector machines (SVM) [65, 66], deep neural networks [69, 70, 71], and multilayer perceptrons [48, 72] However, all these efforts are focused on accelerating a particular learning algorithm.

Several inspiring works propose accelerator designs for a number of learning algorithms [57, 67, 68]. MAPLE [68, 67] profiles five learning algorithms, identifies their compute-intensive kernels, and devises an accelerator that efficiently executes the kernels. PuDianNao [57] provides an ASIC design that can accelerate seven different learning algorithms. We, on the other hand, delves into the theory of machine learning, identify the theoretical commonalities across a wide range of algorithms, devise an abstraction between hardware and software, and provide a unified framework that generates accelerators.

**FPGA as an acceleration platform.** FPGAs have gained popularity due to their flexibility and capability to provide high execution performance by exploiting copious fine-grained irregular parallelism in the applications. Several works [78, 77, 60, 62, 66, 65, 79, 80, 81] utilize FPGAs to accelerate a diverse set of workloads, validating the efficacy of FPGAs. LINQits [82] provides a template architecture for accelerating database queries. The work by King et al. [83] uses Bluespec to automatically generate a hardware-software interface for the applications partitioned for hardware acceleration and software execution. The work by Putnam et al. [4], designs an FPGA fabric for accelerating ranking algorithms in the Bing server. This FPGA-based fabric in deployed with 1632 servers. TABLA provides an opportunity to utilize this integrated reconfigurable fabric for machine learning algorithms. Conclusively, TABLA is a comprehensive solution–from programming language down to circuit design–that provides a unified abstraction based on the theory of machine learning for accelerating an important class of learning algorithms.

## 2.9 Conclusion

Machine learning algorithms are compute-intensive workloads that can benefit significantly from acceleration. FPGAs are an attractive platform for accelerating these important applications. However, FPGA design still requires relatively long development cycles and extensive expertise in hardware design. This paper described TABLA that aims to bridge the gap between the machine learning algorithms and the FPGA accelerators. TABLA dives into the theory of machine learning and takes advantage of the insight that a large class of learning algorithms can be expressed as stochastic optimization problems. TABLA leverages stochastic gradient descent as the abstraction between hardware and software to automatically generate accelerators for this class of statistical machine learning algorithms. We used TABLA to generate accelerators for a variety of learning algorithms targeting an off-the-shelf FPGA platform, Xilinx Zynq. In comparison to a multicore Intel Xeon with vector execution, the TABLA-generated accelerators deliver an average speedup of 2.9×. Compared with the high-performance Tesla K40 GPU accelerator, TABLA achieves 33.4× higher Performance-per-Watt. These gains are achieved while the programmers write less than 50 lines of code. These results suggest that TABLA takes an effective step towards making FPGAs widely available to the machine learning developers. We have made TABLA publicly available (`http://act-lab.org/artifacts/tabla/`) in order to facilitate research and development in using FPGAs for learning.

# CHAPTER 3
# INTEGRATING FULL-STACK ACCELERATION SOLUTIONS WITHIN DATABASE MANAGEMENT SYSTEMS

## 3.1 Summary

The data revolution is fueled by advances in machine learning, databases, and hardware design. Programmable accelerators are making their way into each of these areas independently. As such, there is a void of solutions that enables hardware acceleration at the intersection of these disjoint fields. This paper sets out to be the initial step towards a unifying solution for in-**D**atabase **A**cceleration of Adva**n**ced **A**nalytics (DAnA). Deploying specialized hardware, such as FPGAs, for in-database analytics currently requires hand-designing the hardware and manually routing the data. Instead, DAnA automatically maps a high-level specification of advanced analytics queries to an FPGA accelerator. The accelerator implementation is generated for a User Defined Function (UDF), expressed as a part of an SQL query using a Python-embedded Domain-Specific Language (DSL). To realize an efficient in-database integration, DAnA accelerators contain a novel hardware structure, *Striders*, that directly interface with the buffer pool of the database. *Striders* extract, cleanse, and process the training data tuples that are consumed by a multi-threaded FPGA engine that executes the analytics algorithm. We integrate DAnA with PostgreSQL to generate hardware accelerators for a range of real-world and synthetic datasets running diverse machine lerning algorithms. Results show that DAnA-enhanced PostgreSQL provides, on average, $8.3\times$ end-to-end speedup for real datasets, with a maximum of $28.2\times$. Moreover, DAnA-enhanced PostgreSQL is, on average, $4.0\times$ faster than the multi-threaded Apache MADLib running on Greenplum. DAnA provides these benefits while hiding the complexity of hardware design from data scientists and allowing them to express the algorithm in

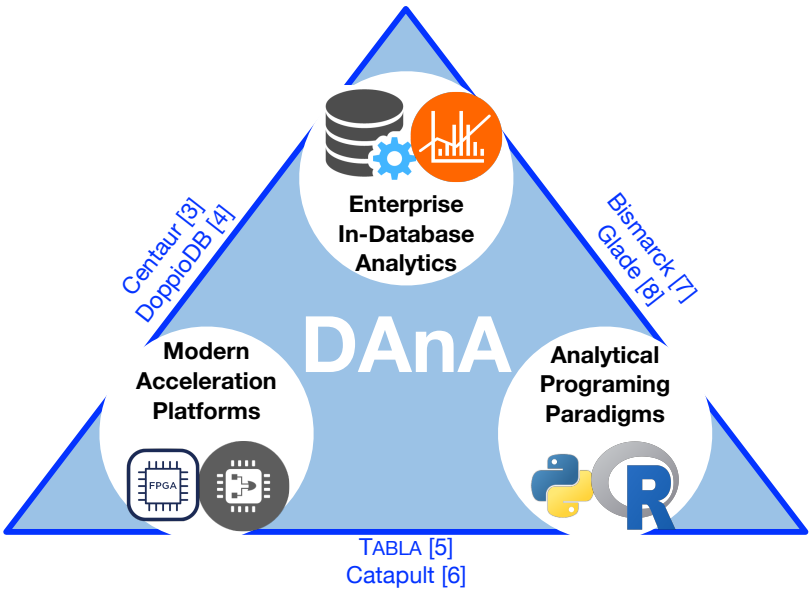



**Figure 3.1:** DAnA represents the fusion of three research directions, in contrast with prior works [84, 85, 12, 78, 19, 38] that merge two of the areas.

≈30-60 lines of Python.

## 3.2 Introduction

Relational Database Management Systems (RDBMSs) are the cornerstone of large-scale data management in almost all major enterprise settings. However, data-driven applications in such environments are increasingly migrating from simple SQL queries towards advanced analytics, especially machine learning, over large datasets [30, 31]. As illustrated in Figure 3.1, there are three concurrent and important, but hitherto disconnected, trends in this data systems landscape: (1) enterprise in-database analytics [84, 85] , (2) modern hardware acceleration platforms [12, 4], and (3) programming paradigms which facilitate the use of analytics [19, 38].

The database industry is investing in the integration of machine learning algorithms within RDBMSs, both on-premise and cloud-based [28, 29]. This integration enables enterprises to exploit machine learning without sacrificing the auxiliary benefits of an RDBMS, such as transparent scalability, access control, security, and integration with their business

intelligence interfaces [32, 33, 34, 35, 36, 19, 37, 38, 39]. Concurrently, the computer architecture community is extensively studying the integration of specialized hardware accelerators within the traditional compute stack for machine learning applications [12, 78, 86, 87, 57]. Recent work at the intersection of databases and computer architecture has led to a growing interest in hardware acceleration for relational queries as well. This includes exploiting GPUs [88] and reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs) [84, 85, 89, 90, 91], for relational operations. Furthermore, cloud service providers like Amazon AWS [6], Microsoft Azure [92], and Google Cloud [5], are also offering high-performance specialized platforms due to the potential gains from modern hardware. Finally, the applicability and practicality of both in-database analytics and hardware acceleration hinge upon exposing a high-level interface to the user. This triad of research areas are currently studied in isolation and are evolving independently. Little work has explored the impact of moving analytics within databases on the design, implementation, and integration of hardware accelerators. Unification of these research directions can help mitigate the inefficiencies and reduced productivity of data scientists who can benefit from in-database hardware acceleration for analytics. Consider the following example.

**Example 1** *A marketing firm uses the Amazon Web Services (AWS) Relational Data Service (RDS) to maintain a PostgreSQL database of its customers. A data scientist in that company forecasts the hourly ad serving load by running a multi-regression model across a hundred features available in their data. Due to large training times, she decides to accelerate her workload using FPGAs on Amazon EC2 F1 instances [6]. Currently, this requires her to learn a hardware description language, such as Verilog or VHDL, program the FPGAs, and go through the painful process of hardware design, testing, and deployment, individually for each machine learning algorithm. Recent research has developed tools to simplify FPGA acceleration for machine learning algorithms [93, 94, 86]. However, these solutions do not interface with or support RDBMSs, requiring her to manually extract, copy, and reformat her large dataset.*

To overcome the aforementioned roadblocks, we devise DAnA, a *cohesive stack that enables deep integration between FPGA acceleration and in-RDBMS execution of advanced analytics*. DAnA exposes a high-level programming interface for data scientists/analysts based on conventional languages, such as SQL and Python. Building such a system requires: (1) providing an intuitive programming abstraction to express the combination of machine learning algorithm and required data schemas; and (2) designing a hardware mechanism that transparently connects the FPGA accelerator to the database engine for direct access to the training data pages.

To address the first challenge, DAnA enables the user to express RDBMS User-Defined Functions (UDFs) using familiar practices of Python and SQL. The user provides their machine learning algorithm as an update rule using a Python-embedded Domain Specific Language (DSL), while an SQL query specifies data management and retrieval. To convert this high level machine learning specification into an accelerated execution without manual intervention, we develop a comprehensive stack. Thus, DAnA is a solution that breaks the algorithm-data pair into software execution on the RDBMS for data retrieval and hardware acceleration for running the analytics algorithm.

With respect to the second challenge, DAnA offers *Striders*, which avoid the inefficiencies of conventional Von-Neumann CPUs for data handoff by seamlessly connecting the RDBMS and FPGA. *Striders* directly feed the data to the analytics accelerator by walking the RDBMS buffer pool. Circumventing the CPU alleviates the cost of data transfer through the traditional memory subsystem. These *Striders* are backed with an Instruction Set Architecture (ISA) to ensure programmability and ability to cater to the variations in the database page organization and tuple length across different algorithms and training datasets. They are designed to ensure *multi-threaded acceleration* of the learning algorithm to amortize the cost of data accesses across concurrent threads. DAnA automatically generates the architecture of these accelerator threads, called execution engines, that selectively combine a Multi-Instruction Multi-Data (MIMD) execution model with Single-Instruction Multi-Data

(SIMD) semantics to reduce the instruction footprint. While generating this MIMD-SIMD accelerator, DAnA tailors its architecture to the machine learning algorithm's computation patterns, RDBMS page format, and available FPGA resources. As such, this paper makes the following technical contributions:

(1) Merges three disjoint research areas to enable transparent and efficient hardware acceleration for in-RDBMS analytics. Data scientists with no hardware design expertise can use DAnA to harness hardware acceleration without manual data retrieval and extraction whilst retaining familiar programming environments.

(2) Exposes a high-level programming interface, which combines SQL UDFs with a Python DSL, to jointly specify training data and computation. This unified abstraction is backed by an extensive compilation workflow that automatically transforms the specification to an accelerated execution.

(3) Integrates an FPGA and an RDBMS engine through *Striders* that are a novel on-chip interfaces. *Striders* bypass CPU to directly access the training data from the buffer pool, transfer this data onto the FPGA, and unpack the feature vectors and labels.

(4) Offers a novel execution model that fuses thread-level and data-level parallelism to execute the learning algorithm computations. This model exposes a domain specific instruction set architecture that offers automation while providing efficiency.

We prototype DAnA with PostgreSQL to automatically accelerate the execution of several popular machine learning algorithms. Through a comprehensive experimental evaluation using real-world and synthetic datasets, we compare DAnA against the popular in-RDBMS ML toolkit, Apache MADlib [35], on both PostgreSQL and its parallel counterpart, Greenplum. Using Xilinx UltraScale+ VU9P FPGA, we observe DAnA generated accelerators provide on average $8.3\times$ and $4.0\times$ end-to-end runtime speedups over PostgreSQL and Greenplum running MADlib, respectively. An average of $4.6\times$ of the speedup benefits are obtained through *Striders*, as they effectively bypass the CPU and its memory subsystem overhead.

### 3.2.1 Insights Driving DAnA

**Database and hardware interface considerations.** To obtain large benefits from hardware acceleration, the overheads of a traditional Von-Neumann architecture and memory subsystem need to be avoided. Moreover, data accesses from the buffer pool need to be at large enough granularities to efficiently utilize the FPGA bandwidth. DAnA satisfies these criteria through *Striders*, its database-aware reconfigurable memory interface, discussed in Section 3.5.1.

**Algorithmic considerations.** The training data retrieved from the buffer pool and stored on-chip must be consumed promptly to avoid throttling the memory resources on the FPGA. DAnA achieves this by leveraging the algorithmic properties of iterative optimization to execute multiple instances of the update rule. The Python-embedded DSL provides a concise means of expressing this update rule for a broad set of algorithms while facilitating parallelization.

DAnA leverages these insights to provide a cross-stack solution that generates FPGA-synthesizable accelerators that directly interface with the RDBMS engine's buffer pool. The next section provides an overview of DAnA.

## 3.3 DAnA Workflow

Figure 3.2 illustrates DAnA's integration within the traditional software stack of data management systems. With DAnA, the data scientist specifies her desired machine learning algorithm as a UDF using a simple DSL integrated within Python. DAnA performs static analysis and compilation of the Python functions to program the FPGA with a high-performance, energy-efficient hardware accelerator design. The hardware design is tailored to both the machine learning algorithm and page specifications of the RDBMS engine. To run the hardware accelerated UDF on her training data, the user provides a SQL query.

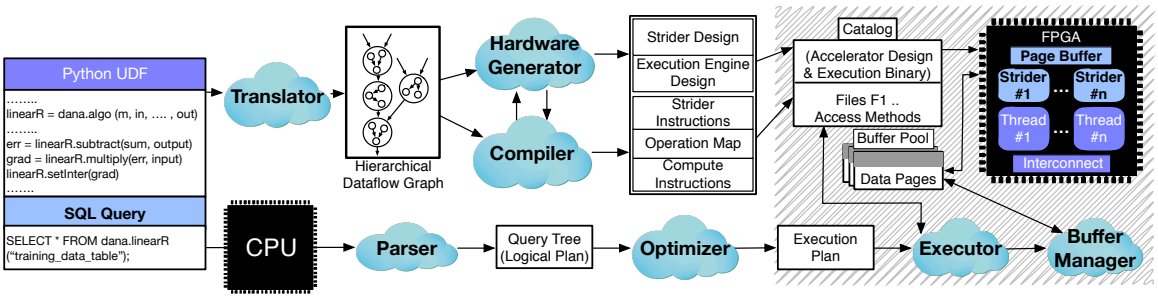



**Figure 3.2:** Overview of DAnA, that integrates FPGA acceleration with the RDBMS engine. The Python-embedded DSL is an interface to express the machine learning algorithm that is converted to hardware architecture and its execution schedules (stored in the RDBMS catalog). The RDBMS engine fills the buffer pool. FPGA *Striders* directly access the data pages to extract the tuples and feed them to the threads. Shaded areas show the entangled components of RDBMS and FPGA working in tandem to accelerate in-database analytics.

DAnA stores accelerator metadata (*Strider* and execution engine instruction schedules) in the RDBMS's catalog along with the name of a UDF to be invoked from the query. As shown in Figure 3.2, the RDBMS catalog is shared by the database engine and the FPGA. The RDBMS parses, optimizes, and executes the query while treating the UDF as a black box. During query execution, the RDBMS fills the buffer pool, from which DAnA ships the data pages to the FPGA for processing. DAnA and the RDBMS engine work in tandem to generate the appropriate data stream, data route, and accelerator design for the {machine learning algorithm, database page layout, FPGA} triad. Each component of DAnA is briefly described below.

**Programming interface.** The front end of DAnA exposes a Python-embedded DSL (discussed in Section 3.4.1) to express the machine learning algorithm as a UDF. The UDF includes an update rule that specifies how each tuple or record in the training data updates the model. It also expects a merge function that specifies how to process multiple tuples in parallel and aggregate the resulting machine learning models. DAnA's DSL constitutes a diverse set of operations and data types that cater to a wide range of advanced analytics algorithms. Any legitimate combination of these operations can be automatically converted to a final synthesizable FPGA accelerator.

**Translator.** The user provided UDF is converted into a *h*ierarchical DataFlow Graph

($h$DFG) by DAnA's parser, discussed in detail in Section 3.4.4. Each node in the $h$DFG represents a mathematical operation allowed by the DSL, and each edge is a multi-dimensional vector on which the operations are performed. The information in the $h$DFG enables DAnA's backend to optimally customize the reconfigurable architecture and schedule and map each operation for a high-performance execution.

**Strider-based customizable machine learning architecture.** To target a wide range of machine learning algorithms, DAnA offers a parametric reconfigurable hardware design solution that is hand optimized by expert hardware designers as described in Section 3.5. The hardware interfaces with the database engine through a specialized structure called *Striders*, that extract high-performance, and provide low-energy computation. *Striders* eliminate CPU from the data transformation process by directly interfacing with database's buffer pool to extract the training data pages. They process data at a page granularity to amortize the cost of per-tuple data transfer from memory to the FPGA. To exploit this vast amount of data available on-chip, the architecture is equipped with execution engines that run multiple parallel instances of the update rule. This architecture is customized by DAnA's compiler and hardware generator in accordance to the FPGA specifications, database page layout, and the analytics function.

**Instruction Set Architectures.** Both *Striders* and the execution engine can be programmed using their respective Instruction Set Architectures (ISAs). The *Strider* instructions process page headers, tuple headers, and extract the raw training data from a database page. Different page sizes and page layouts can be targeted using this ISA. The execution engine's ISA describes the operation flow required to run the analytics algorithm in selective SIMD mode.

**Compiler and hardware generator.** DAnA's compiler and hardware generator ensure compatibility between the $h$DFG and the hardware accelerator. For the given $h$DFG and FPGA specifications (such as number of DSP Slices and BRAMs), the hardware generator determines the parameters for the execution engine and *Striders* to generate the final FPGA

synthesizable accelerator. The compiler converts the database page configuration into a set of *Strider* instructions that process the page and tuple headers and transform user data into a floating point format. Additionally, the compiler generates a static schedule for the accelerator, a map of where each operation is performed, and execution engine instructions. As described above, providing flexibility and reconfigurability of hardware accelerators for advanced analytics is a challenging but pertinent problem. DAnA is a multifaceted solution that untangles these challenges one by one.

## 3.4 Front-End Interface of DAnA

DAnA's DSL provides an entry point for data scientists to exploit hardware acceleration for in-RDBMS advanced analytics. This section elaborates on the constructs and features of the DSL and how they can be used to train a wide range of learning algorithms for advanced analytics. This section also explains how a UDF defined in this DSL is translated into an intermediate representation, i.e., in this case a $h$ierarchical DataFlow Graph ($h$DFG).

### 3.4.1 Programming For DAnA

DAnA exposes a high-level DSL for database users to express their learning algorithm as a UDF. Embedding this DSL within Python allows support for intricate update rules using a framework familiar to database users whilst not requiring a full language compiler. This DSL meets the following objectives:

- Incorporates language constructs commonly seen in a wide class of supervised learning algorithms.
- Supports expression of any iterative update rule, not just variants of gradient descent, whilst conforming to the DSL constructs.
- Segregates algorithmic specification from hardware-dependent implementation.

The language constructs of this DSL – *components*, *data type qualifiers*, *mathematical*

**Table 3.1:** Language constructs of DAnA's Python-embedded DSL.

| Type | Keyword | Description |
|---|---|---|
| **Component** | algo | To specify an instance of the learning algorithm |
| **Data Types** | input | Algorithm input |
| | output | Algorithm output |
| | model | Machine learning model |
| | inter | Interim data type |
| | meta | Meta parameters |
| **Mathematical Operations** | +,-,*, /, >, < | Primary operations |
| | sigmoid, gaussian, sqrt | Non linear operations |
| | sigma, norm, pi | Group operations |
| **Built-In Special Functions** | merge(x, int, "operation") | Specify merge operation and number of merge instances |
| | setEpochs(int) | Set the maximum number of epochs |
| | setConvergence(x) | Specify the convergence criterion |
| | setModel(x) | Set the model variable |

*operations*, and *built-in functions* – are summarized in Table 3.1. Users express the learning algorithm using these constructs and provide the (1) *update rule* - to decide how each tuple in the training data updates the model; (2) *merge function* - to specify the combination of distinct parallel update rule threads; and (3) *terminator* - to describe convergence.

### 3.4.2 Language Constructs

**Data type qualifiers.** Data declarations delineate the semantics of the data types used in the machine learning algorithm. The DSL supports the following data declarations: input, output, inter, model, and meta. Each variable can be declared by specifying its type and dimensions. A variable is an implied scalar if no dimensions are specified. Once the analyst imports the dana package, she can express the required variables. The code snippet below declares a multi-dimensional machine learning model of size [5][2] using dana.model construct.

```
mo = dana.model ([5][2])
```

In addition to dana.model, the user can provide dana.input and dana.output to express a single input-output pair in the training dataset. The user can specify meta variables using

dana.meta, the value of which remains constant throughout execution. As such, meta variables can be directly sent to the FPGA before algorithm execution. All variables used for a particular algorithm are linked to an algo construct.

```
algorithm = dana.algo (mo, in, out)
```

The algo component allows the user to link together the three functions – update rule, merge, and terminator – of a single UDF. Additionally, the analyst can use untyped intermediate variables, which are automatically labeled as dana.inter by DAnA's backend.

**Mathematical operations.** The DSL supports mathematical operations performed on both declared and untyped intermediate variables. Primary and non-linear operations, such as *, +, ... , sigmoid, only require the operands as input. The dimensionality of the operation and its output is automatically inferred by DAnA's translator (as discussed in Section 3.4.4) in accordance to the operands' dimensions. Group operations, such as sigma, pi, norm, perform computation across elements. Sigma refers to summation, pi indicates product operator, and norm calculates the magnitude of a multidimensional vector. Group operations require the input operands and the grouping axis which is expressed as a constant and alleviates the need to explicitly specify loops. The underlying primary operation is performed on the input operands prior to grouping.

**Built-in functions.** The DSL provides four built-in functions to specify the merge condition, set the convergence criterion, and link the updated model variable to the algo component. The merge(x, int, "op") function is used to specify how multiple threads of the update rule are combined. Convergence is dictated either by a specifying fixed number of epochs (1 epoch is a single pass over the entire training data set) or a user-specified condition. Function setEpochs(int) sets the number of terminating epochs and setConvergence(x) frames termination based on a boolean variable x. Finally, the setModel(x) function links a DAnA variable (the updated model) to the corresponding algo component.

All the described language constructs are supported by DAnA's reconfigurable archi-

tecture, hence, can be synthesized on an FPGA. An example usage of these constructs to express the update rule, merge function, and convergence for linear regression algorithm running the gradient descent optimizer is provided below.

### 3.4.3 Linear Regression Example

**Update rule.** As the code snippet below illustrates, the data scientist first declares different data types and their corresponding dimensions. Then she defines the computations performed over these variables specific to linear regression.

```
#Data Declarations
mo = dana.model ([10])
in = dana.input ([10])
out  = dana.output ()
lr = dana.meta (0.3) #learning rate

linearR = dana.algo (mo, in, out)

#Gradient or Derivative of the Loss Function
s = sigma ( mo * in, 1)
er = s - out
grad = er * in

#Gradient Descent Optimizer
up = lr * grad
mo_up = mo - up
linearR.setModel(mo_up)
```

In this example, the update rule uses the gradient of the loss function. The gradient descent optimizer updates the model in the negative direction of the loss function derivative ($\frac{\partial(l)}{\partial w^{(t)}}$). The analyst concludes with the setModel() function to identify the updated model, in this case mo_up.

**Merge function.** The merge function facilitates multiple concurrent threads of the update rule on the FPGA accelerator by specifying the functionality at the point of merge.

```
merge_coef = dana.meta (8)
grad = linearR.merge(grad, merge_coef, "+")
```

In the above merge function, the intermediate grad variable has been combined using addition, and the merge coefficient (merge_coef) specifies the batch size. DAnA's compiler implicitly understands that the merge function is performed before the gradient descent optimizer. Specifically, the grad variable is calculated separately for each tuple per batch. The results are aggregated together across the batches and used to update the model. Alternatively, partial model updates for each batch could be merged.

```
merge_coef = dana.meta (8)
m1 = linearR.merge(mo_up, merge_coef, "+")
m2 = m1/merge_coef
lineaR.setModel(m2)
```

The mo_up is calculated by each thread for tuples in its batch separately and consecutively averaged. Thus, DAnA's DSL provides the flexibility to create different learning algorithms without requiring any modification to the update rule by specifying different merge points. In the above example, the first definition of the merge function creates a linear regression running batched gradient descent optimizer, whereas, the second definition corresponds to a parallelized stochastic gradient descent optimizer.

**Convergence function.** The user also provides the termination criteria. As shown in the code snippet below, the convergence checks for the conv variable, which, if true, terminates the training. Variable conv compares the Euclidean norm of grad with a conv_factor constant.

```
convergenceFactor = dana.meta (0.01)
n = norm(grad , i)
conv = n < convergenceFactor
linear.setConvergence(conv)
```

Alternatively, the number of epochs can be used for convergence using the syntax linearR.setEpochs(10000).

**Query.** A UDF comprising the update rule, merge function, and convergence check describes the entire analytics algorithm. The linearR UDF can then be called within a query as follows:

```
SELECT * FROM dana.linearR('training_data_table');
```

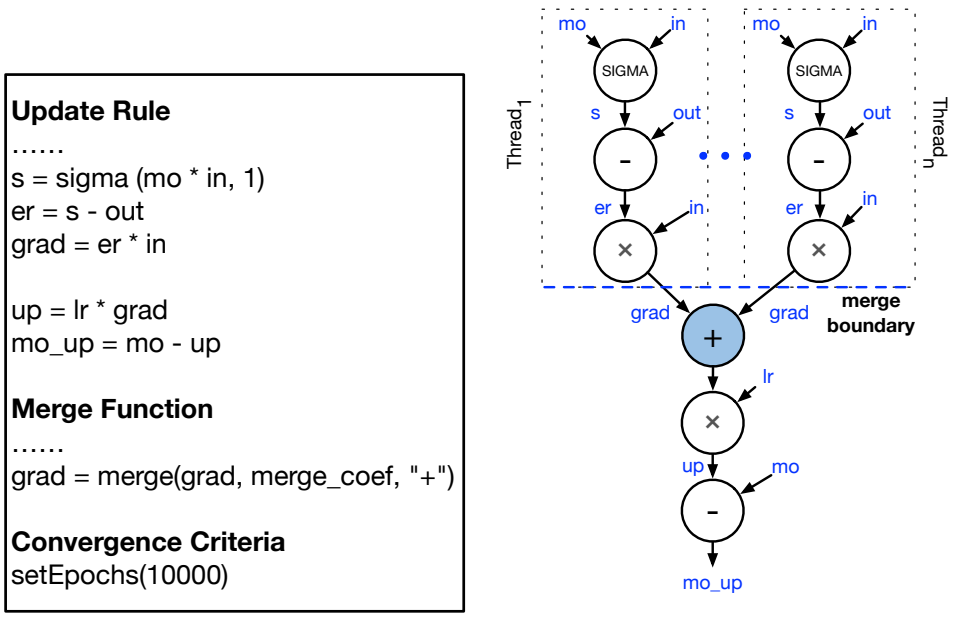
Currently, for high efficiency and low latency, DAnA's DSL and compiler do not support dynamic variables, as the FPGA and CPU do not interchange runtime values and only interact for data handoff. DAnA only supports variable types which either have been explicitly instantiated as DAnA's data declarations, or inferred as intermediate variables (dana.inter) by DAnA's translator. As such, this Python-embedded DSL provides a high level programming abstraction that can easily be invoked by an SQL query and extended to incorporate algorithmic advancements. In the next section we discuss the process of converting this UDF into a $h$DFG.

### 3.4.4 Translator

DAnA's translator is the front-end of the compiler, which converts the user-provided UDF to a $h$ierarchical DataFlow Graph ($h$DFG). The $h$DFG represents the coalesced update rule, merge function, and convergence check whilst maintaining the data dependencies. Each node of the $h$DFG represents a multi-dimensional operation, which can be decomposed into smaller atomic sub-nodes. An atomic sub-node is a single operation performed by the accelerator. The $h$DFG transformation for the linear regression example provided in the previous section is shown in Figure 3.3.

The aim of the translator is to expose as much parallelism available in the algorithm to the remainder of the DAnA workflow. This includes parallelism within a single instance of the update rule and among different threads, each running a version of the update rule. To accomplish this, the translator (1) maintains the function boundaries, especially between the merge function and parallelizable portions of the update rule, and (2) automatically infers dimensionality of nodes and edges in the graph.

The merge function and convergence criteria are performed once per epoch. In Figure 3.3b, the colored node represents the merge operation that combines the gradients generated by separate instances of the update rule. These update rule instances are run




**(a)** Code snippet **(b)** Hierarchical DFG

**Figure 3.3:** Translator-generated $h$DFG for the linear regression code snippet expressed in DAnA's DSL.

in parallel and consume different records or tuples from the training data; thus, they can be readily parallelized across multiple threads. To generate the $h$DFG, the translator first infers the dimensions of each operation node and its output edge(s). For basic operations, if both the inputs have same dimensions, it translates into an element by element operation in the hardware. In case the inputs do not have same dimensions, the input with lower dimension is logically replicated, and the generated output possess the dimensions of the larger input. Nonlinear operations have a single input that determines the output dimensions. For group operations, the output dimension is determined by the axis constant. For example, a node performing sigma(mo * in, 2), where variables mo and in are matrices of sizes [5][10] and [2][10], respectively, generates a [5][2] output.

The information captured within the $h$DFG allows the hardware generator to configure the accelerator architecture to optimally cater for its operations. Resources available on the FPGA are distributed on-demand within and across multiple threads. Furthermore, DAnA's compiler maps all the operations to the accelerator architecture to exploit fine-grained par-

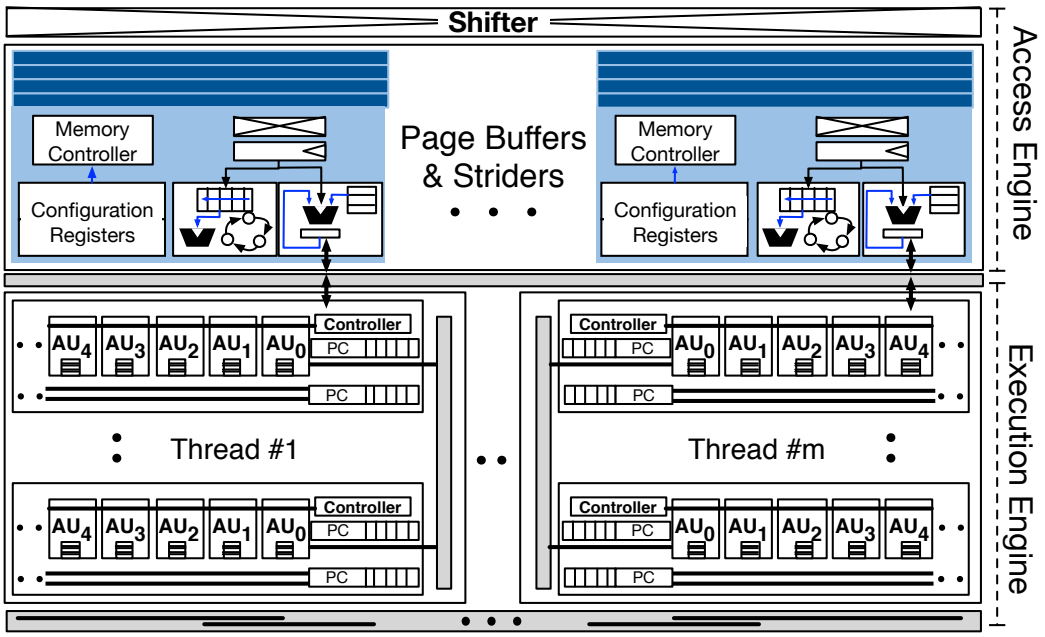



**Figure 3.4:** Reconfigurable accelerator design in its entirety. The access engine reads and processes the data via its *Striders*, while the execution engine operates on this data according to the UDF.

allelism within an update rule. Before delving into the details of hardware generation and compilation, we discuss the reconfigurable architecture for the FPGA (*Strider* and execution engine).

## 3.5 Hardware Design for in-Database Acceleration

DAnA employs a parametric accelerator architecture comprising a *multi-threaded access engine* and a *multi-threaded execution engine*, shown in Figure 3.4. Both engines have their respective custom Instruction Set Architectures (ISA) to program their hardware designs. The access engine harbors *Striders* to ensure compatibility between the data stored in a particular database engine and the execution engines that perform the computations required by the learning algorithm. The access and execution engines are configured according to the page layout and UDF specification, respectively. The details of each of these components are discussed below.

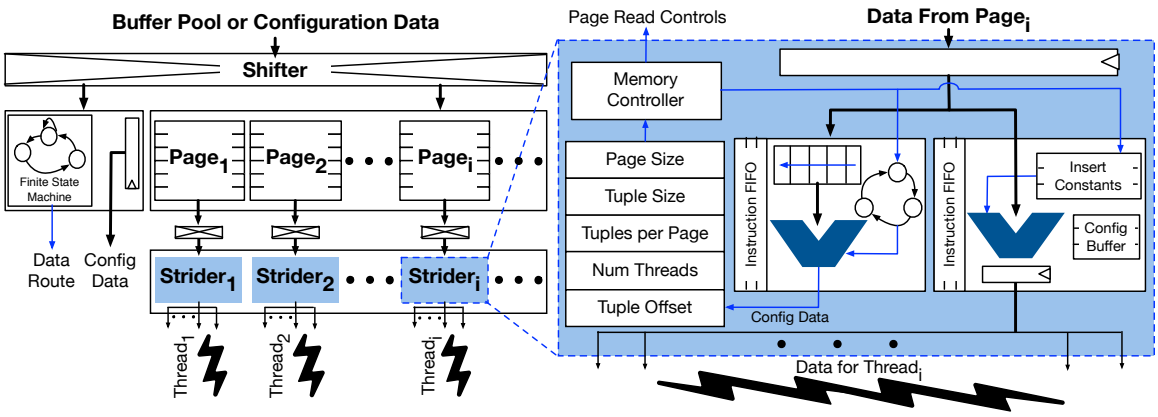



**Figure 3.5:** Access engine design uses *Striders* as the main interface between the RDBMS and execution engines. Uncompressed data pages are read from the buffer pool and stored in on-chip page buffers. Each page has a corresponding strider to extract the tuple data.

## 3.5.1 Access Engine and Striders

### *3.5.1.1 Architecture and Design*

The multi-threaded access engine is responsible for storing pages of data and converting them from a database page format to raw numbers that are processed by the execution engine. Figure 3.5 shows a detailed diagram of this access engine. The access engine uses the Advanced Extensible Interface (AXI) interface to transfer the data to and from the FPGA, the shifters properly align the data, and the *Striders* unpack the database pages. AXI interface is a type of Advanced Microcontroller Bus Architecture open-standard, on-chip interconnect specification for system-on-a-chip (SoC) designs. It is vendor agnostic and standardized across different hardware platforms. The access engine uses this interface to transfer uncompressed database pages to page buffers and configuration data to configuration registers. Configuration data comprises *Strider* and execution engine instructions and necessary meta-data. Both the training data in the database pages and the configuration data are passed through a shifter for alignment, according to the read width of the block RAM on the target FPGA. A separate channel for configuration data incorporates a finite state machine to dictate the route and destination of the configuration information.

To amortize the cost of data transfer and avoid the suboptimal usage of the FPGA

bandwidth, the access engine and *Striders* process database data at a page level granularity. Training data is written to multiple page buffers, where each buffer stores one database page at a time and has access to its personal *Strider*. Alternatively, each tuple could have been extracted from the page by the CPU and sent to the FPGA for consumption by the execution engine. This approach would fail to exploit the bandwidth available on the FPGA, as only one tuple would be sent at a time. Furthermore, using the CPU for data extraction would have a significant overhead due to the handshaking between CPU and FPGA. Offloading tuple extraction to the accelerator using *Striders* provides a unique opportunity to dynamically interleave unpacking of data in the access engine and processing it in the execution engine.

It is common for data to be spread across pages, where each page requires plenty of pointer chasing. Two tuples cannot be simultaneously processed from a single page buffer, as the location of one could depend on the previous. Therefore, we store multiple pages on the FPGA and parallelize data extraction from the pages across their corresponding *Striders*. For every page, the *Strider* first processes the page header and extracts necessary information about the page and stores it in the configuration registers. The information includes offsets, such as the beginning and size of each tuple, which is either located or computed from the data in the header. This auxiliary page information is used to trace the tuple addresses and read the corresponding data from the page buffer. After each page buffer, the shifter ensures alignment of the tuple data for the *Strider*. From the tuple data, its header is processed to extract and route the training data to the execution engine. The number of *Striders* and database pages stored on-chip can be adjusted according to the BRAM storage available on the target FPGA. The internal workings of the *Strider* are dictated by its instructions that depend on the page layout and page size of the target RDBMS. We next discuss the novel ISA to program these *Striders*.

**Table 3.2:** *Strider* ISA to read, extract, and clean the page data.

| Instruction | Instruction Code | Bits | | | |
|---|---|---|---|---|---|
| | | 21 - 18 | 17 - 12 | 11 - 6 | 5 - 0 |
| Read Bytes | readB | Opcode = 0 | Read Address | | |
| Extract Bytes | extrB | Opcode = 1 | Byte Offset | | |
| Write Bytes | writeB | Opcode = 2 | Read Address | # of Bytes | Write Address |
| Extract Bits | extrBi | Opcode = 3 | | | |
| Clean | cln | Opcode = 4 | | | # of Bits |
| Insert | ins | Opcode = 5 | Start Location | Offset | Reserved |
| Add | ad | Opcode = 6 | | | |
| Subtract | sub | Opcode = 7 | | | Immediate |
| Multiply | mul | Opcode = 8 | Read Address 1 | Read Address 2 | Operand |
| Branch Enter | bentr | Opcode = 9 | 0 | | |
| Branch Exit | bexit | Opcode = 10 | Condition | Value | |

### *3.5.1.2 Instruction Set Architecture for* Striders

We devise a novel fixed-length Instruction Set Architecture (ISA) for the *Striders* that can target a range of RDBMS engines, such as PostgreSQL and MySQL (innoDB), that have similar backend page layouts. An uncompressed page from these RDBMSs, once transferred to the page buffers, can be read, extracted, and cleansed using this ISA, which comprises light-weight instructions specialized for pointer chasing and data extraction. Each *Strider* is programmed with the same instructions but operates on different pages. These instructions are generated statically by the compiler.

Table 3.2 illustrates the 10 instructions of this ISA. Every instruction is 22 bits long, comprising a unique operation code (opcode) as identification. The remaining bits are specific to the opcode. Instructions **Read Bytes** and **Write Bytes** are responsible for reading and writing data from the page buffer, respectively. The ISA provides the flexibility to extract data at byte and bit granularity using the **Extract Byte** and **Extract Bit** instructions. The **Clean** instruction can remove parts of the data not required by the execution engine. Conversely, the **Insert** instruction can add bits to the data, such as NULL characters and auxiliary information, which is particularly useful when the page is to be written back to memory. Basic math operations, **Add**, **Subtract**, and **Multiply**, allow calculation of tuple

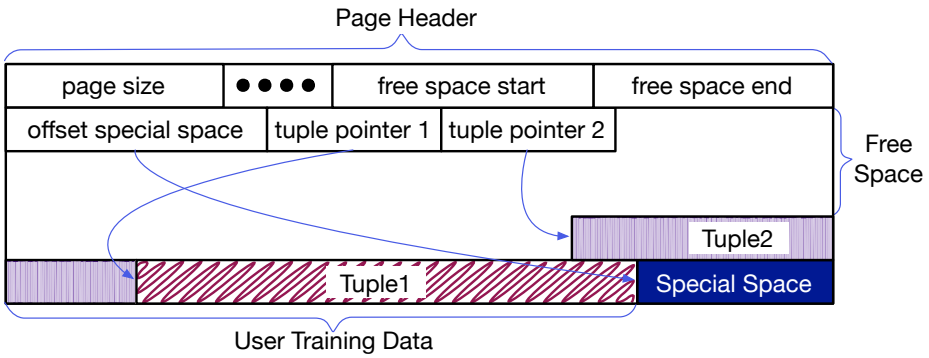



**Figure 3.6:** Sample page layout similar to PostgreSQL.

sizes, byte offsets, etc. Finally, the **Bentr** and **Bexit** branch instructions are used to specify jumps or loop exits, respectively. This feature invariably reduces the instruction footprint as repeated patterns can be succinctly expressed using branches while enabling loop exits that depend on a dynamic runtime variable.

An example page layout representative of PostgreSQL and MySQL is illustrated in Figure 3.6. Such layouts are divided into a page header, tuple pointers, and tuple data and can be processed using the following assembly code snippet written in *Strider* ISA.

```
\\Page Header Processing
readB 0, 8, %cr
readB 8, 2, %cr
readB 10, 4, %cr
extrB %cr, 2, %cr

\\Tuple Pointer Processing
readB %cr, 4, %treg
extrB 0, 1 ,%cr
extrB 1, 1 ,%treg

\\Tuple extraction and processing
bentr
ad %treg, %treg, 0
readB %treg, %cr, %treg
extrB %treg, %cr, %treg
cln %treg, %cr, 2
bexit 1, %treg, %cr
```

Each line in the assembly code is identified by its instruction name (opcode) and its corresponding fields. The first four assembly instructions process the page header to obtain

the configuration information. For example, the **(readB 0, 8, %cr)** instruction, reads 8 bytes from address 0 in the page buffer and adds this page size information into a configuration register. Each variable shown at %(reg) corresponds to an actual *Strider* hardware register. The %cr is a configuration register, and %t is a temporary register. Next, the first tuple pointer is read to extract the byte-offset and length (bytes) of the tuple. Only the first tuple pointer is processed, as all the training data tuples are expected to be identical. Each corresponding tuple is processed by adding the tuple size to the previous offset to generate the page address. This address is used to read the data from the page buffer, which is then cleansed by removing its auxiliary information. The above step is repeated for each tuple using the **bentr** and **bexit** instructions. The loop is exited when the tuple offset address reaches the free space in the page. Finally, cleaned data is sent to the execution engines.

### 3.5.2 Execution Engine

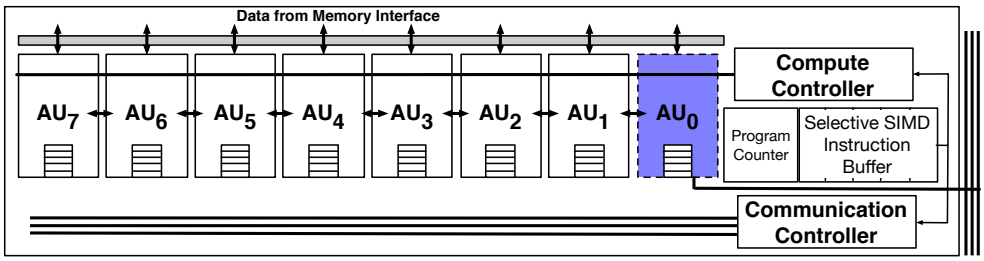
The execution engines execute the $h$DFG of the user provided UDF using the *Strider* processed training data pages. More and more database pages can now be stored on-chip as the BRAM capacity is rapidly increasing with the new FPGAs such as Arria 10 that offers 7 MB and UltraScale+ VU9P with 44 MB of memory. Therefore, the execution engine needs to furnish enough computational resources that can process this copious amount of on-chip data. Our reconfigurable execution engine architecture can run multiple threads of parallel update rules for different data tuples. This architecture is backed by a *Variable Length Selective SIMD ISA*, that aims to exploit both regular and irregular parallelism in machine learning algorithms whilst providing the flexibility to each component of the architecture to run independently.

#### *3.5.2.1 Reconfigurable Compute Architecture*

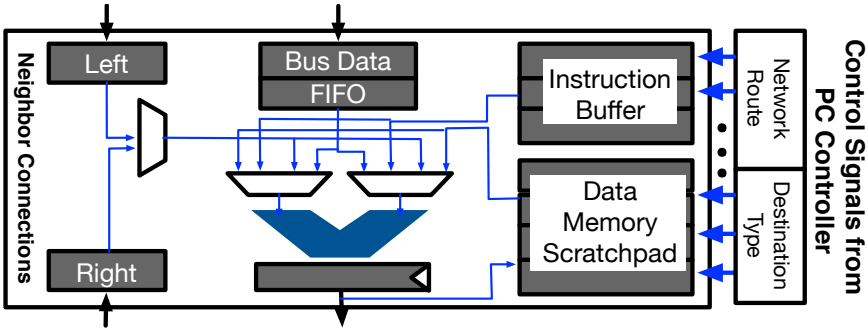
All the threads in the execution engine are architecturally identical and perform the same computations on different training data tuples. DAnA balances the resources allocated

per thread vs. the number of threads to ensure high performance for each algorithm. The hardware generator of DAnA (discussed in Section 3.6.1) determines this division by taking into account the parallelism in the $h$DFG, number of compute resources available on chip, and number of striders/page buffers that can fit on the on-chip BRAM. The architecture of a single thread is a hierarchical design comprising analytic clusters (ACs) composed of multiple analytic units (AUs). As discussed below, the AC architecture is designed while keeping in mind the algorithmic properties of multi-threaded iterative optimizations, and the AU caters to commonly seen compute operations in data analytics.

**Analytic cluster.** An Analytic Cluster (AC), shown in Figure 3.7a, is a collection of AUs designed to reduce the data transfer latency between them. Thus, $h$DFG nodes which exhibit high data dependencies are all scheduled to a single cluster. In addition to providing greater connectivity among the AUs within an AC, the cluster serves as the control hub for all its constituent AUs. The AC runs in a selective SIMD mode, where the AC specifies which AUs within a cluster perform an operation. Each AU within a cluster is expected to execute either a cluster level instruction (add, subtract, multiply, divide, etc.) or a no-operation (NOP). Finer details about the source type, source operands, and destination type can be stored in each individual AU for additional flexibility. This collective instruction technique simplifies the AU design, as each AU no longer requires a separate controller to decode and process the instruction. Instead, the AC controller processes the instruction and sends control signals to all the AUs. When the designated AUs complete their execution, the AC proceeds to the next instruction by incrementing the program counter. To exploit the data locality among the operations performed within an AC, different connectivity options are provided. Each AU within an AC is connected to both its neighbors, and the AC has a shared line topology bus. The number of AUs per AC are fixed to 8 to obtain highest operational frequency. A single thread generally contains more than one instance of an AC, each performing instructions independently. Data sharing among ACs is made possible via




**(a)** Analytic Cluster




**(b)** Analytic Unit

**Figure 3.7:** (a) Single analytic cluster comprising analytic units operating in a selective SIMD mode and (b) an analytic unit that is the pipelined compute hub of the architecture.

a shared line topology inter-AC bus.

**Analytic unit.** The Analytic Unit (AU) shown in Figure 3.7b, is the basic compute element of the execution engine. It is tailored by the hardware generator to satisfy the mathematical requirements of the $h$DFG. Control signals are provided by the AC. Data for each operation can be read from the memory according to the source type of the AC instruction. Training data and intermediate results are stored in the data memory. Additionally, data can be read from the bus FIFO (First In First Out) and/or the registers corresponding to the left and right neighbor AUs. Data is then sent to the Arithmetic Logic Unit (ALU), that executes both basic mathematical operations and complicated non-linear operations, such as sigmoid, gaussian, and square root. The internals of the ALU are reconfigured according to the operations required by the $h$DFG. The ALU then sends its output to the neighboring AUs, the shared bus within the AC, and/or the memory as per the instruction.

**Bringing the Execution Engine together.** Results across the threads are combined via a computationally-enabled tree bus in accordance to the merge function. This bus has attached ALUs to perform computations on in-flight data. The pliability of the architecture enables DAnA to generate high-performance designs that efficiently utilize the resources on the FPGA for the given RDBMS engine and algorithm. The execution engine is programmed using its own novel ISA, described in the next section.

#### *3.5.2.2 Instruction Set Architecture for Execution Engine*

Our variable-length ISA for the execution engines supports a Selective SIMD processing model, which targets two types of nodes in the $h$DFG: those easily vectorized and those which exhibit limited parallelism due to high data dependencies. A variable length ISA elongates the decoding process but reduces its overall memory footprint, leaving more room for the data pages to be stored on on-chip. Despite being variable length, every instruction is self-sufficient and contains all relevant material.

As shown in Table 3.3a, this ISA has three instruction types: compute, communication, and operand reads. These are micro-instructions generated for each operation to be performed in the $h$DFG. The blue-shaded fields in the table are mandatory, while the remaining are optional. Fields specified as N/A are not a part of the actual micro-instruction and are fillers to align the fields relevant to each AU for illustration purposes. Each operation to be performed always has the compute and write back/communication micro-instructions stored inside the AC's instruction buffer. In contrast, the operand read micro-instructions are optional and are stored inside the AU's instruction buffer. For the **compute** micro-instructions, the first field is a byte which indicates the operational AUs. This field is required and specifies which AUs perform the operation. The next field is the mathematical operation identifier specified using an opcode. Operations currently supported by DAnA's DSL and hardware are listed in Table 3.3b.

AUs that participate in executing the operation have the corresponding **operand read**

**Table 3.3:** (a) A variable-length ISA for the execution engine, (b) its supported operations, and (c) the types of instruction operands.

**(a)** Three Categories of Instructions

| Instruction Type | | | | PE0 | PE1 | PE2 | PE3 | PE4 | PE5 | PE6 | PE7 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Compute | Operation | Opcode | | Operational PE (1 Byte) | | | | | | | |
| Write Back / Communicate | Global Bus | Use | PE User | Destination PE (1 Byte) | | | | | | | |
| | Local Bus | | | | | | | | | | |
| | Memory | | | Write to Scratchpad (1 Byte) | | | | | | | |
| Operand Read | Op I Type | | | Source Type (1/2 Byte - 2 Bytes) | | | | | .... | | |
| | Op II Type | | | | | | | | .... | | |
| | Op I Index | | | Source Index (2 Bits - n Bytes) | | | | | .... | | |
| | Op II Index | | | | | | | | .... | | |

**(b)** Supported Math

| | | |
|---|---|---|
| Basic Compute Instructions | Add | Opcode = 1 |
| | Subtract | Opcode = 2 |
| | Divide | Opcode = 3 |
| | Multiply | Opcode = 4 |
| Group Compute Instructions | Sum | Opcode = 5 |
| | Norm | Opcode = 6 |
| | Pi | Opcode = 7 |
| Non Linear Compute Instructions | Sigmoid | Opcode = 8 |
| | Gaussian | Opcode = 9 |
| | Sqrt | Opcode = 10 |

**(c)** Operand Types

| Operand Type | Code | Index |
|---|---|---|
| Data Memory | 00 | Address |
| Scratchpad | 01 | |
| Neighbor | 02 | Left/Right |
| Bus | 03 | Local/Global |

micro-instructions in their instruction buffer. These operand read micro-instructions have two types: operand type and operand index. The operand type for any AU specifies the source type of the operation. Figure 3.3c shows all the operand types – data memory, scratchpad, neighbor register, and bus FIFO – from which each AU can read or write its operands or outputs, respectively. The operand index instruction is only valid if the operand is of the data memory type. Finally, the **Communicate** and **Write Back** micro-instructions itemize where the data is to be written by the AUs performing the operation. The first field, Use, is required and specifies whether the write back or communication component is used. For example, if the local bus Use field is 1, the Source AU uses the bus to transfer data to all AUs indicated by the "Destination AU" field. A use case of this ISA is shown with a simple group operation **sigma** shown below. This operation is performed in an AC using operations shown in Table 3.4.

**Table 3.4:** Operation flow in an AC to perform Equation 3.1.

| AU0 | AU1 | AU2 | AU3 | AU4 | AU5 | AU6 | AU7 |
|---|---|---|---|---|---|---|---|
| $*$ | $*$ | $*$ | $*$ | $*$ | $*$ | $*$ | $*$ |
| $+$ | | $+$ | | $+$ | | $+$ | |
| $+$ | | | | $+$ | | | |
| $+$ | | | | | | | |

$$s = sigma(x_i \times w_i, [8]) \tag{3.1}$$

All AUs in an AC perform the multiply operation. The results are aggregated through a reduction tree to generate the final result in AU 0. The first multiply operation is converted to a compute micro-instruction with opcode 100 (from Table 3.3b) and operational AU value of 11111111, as all the AUs perform the operation. The operand read micro-instruction for each AU points towards data memory, encoded as 00 00 for both operands. Operand indices are set to 0000 0001, assuming both the multiplicand and the multiplier are in consecutive memory locations. Neither the global bus nor the local bus is used, therefore, the Use field is 0, and none of the remainder fields in those micro-instructions are required. For AUs 0, 2, 4, and 8, a subsequent add operation needs to be performed on the previous result. Hence, the multiply output needs to be stored in scratchpad via a memory micro-instruction. The Use field corresponding to the scratchpad is set to 1, while the Write to Scratchpad field is set to 10101010 indicating that only alternating AUs write.

These components ensure compatibility between the high level Python UDF and the access and execution engines by generating programs for each.

## 3.6 Compilation Workflow

DAnA's translator, scheduler, and hardware generator together configure the accelerator design for the UDF and create its runtime schedule. As discussed in Section 3.4.4, the translator converts the user-provided UDF, merge function, and convergence criteria into a

$h$DFG. Each node of the $h$DFG comprises of sub-nodes, where each sub-node is a single instruction in the execution engine. Thus, all the sub-nodes in the $h$DFG are scheduled and mapped to the final accelerator hardware design. The hardware generator outputs a single-threaded architecture for the operations of these sub-nodes and determines the number of threads to be instantiated. The scheduler then statically maps all operations to this architecture.

### 3.6.1 Hardware Generator

The hardware generator finalizes the parameters of the reconfigurable architecture (Figure 3.4) for the *Striders* and the execution engine. The hardware generator obtains the database page layout information, model, and training data schema from the DBMS catalog. FPGA-specific information, such as the number of DSP slices, the number of BRAMs, the capacity of each BRAM, the number of read/write ports on a BRAM, and the off-chip communication bandwidth are provided by the user. Using this information, the hardware generator distributes the resources among access and execution engine. Sizes of the DBMS page, model, and a single training data record determine the amount of memory utilized by each *Strider*. Specifically, a portion of the BRAM is allocated to store the extracted raw training data and model. The remainder of the BRAM memory is assigned to the page buffer to store as many pages as possible to maximize the off-chip bandwidth utilization.

Once the number of resident pages is determined, the hardware generator uses the FPGA's DSP information to calculate the number of AUs which can be synthesized on the target FPGA. Within each AU, the ALU is customized to contain all the operations required by the $h$DFG. The number of AUs determines the number of ACs. Each thread is allocated a number of ACs determined by the merge coefficient provided by the programmer. It creates at most as many threads as the coefficient. To decide the allocation of resources to each thread vs. number of threads, we equip the hardware generator with a performance estimation tool that uses the static schedule of the operations for each design

point to estimate its relative performance. It chooses the smallest and best-performing design point which strikes a balance between the number of cycles for data processing and transfer. Performance estimation is viable, as the $h$DFG does not change, there is no hardware managed cache, and the accelerator architecture is fixed during execution. Thus, there are no dynamic irregularities that hinder estimation. This technique is commensurate with prior works [12, 86, 94] that perform a similar restricted design space exploration in less than five minutes with estimates within 5% of the physical measurements.

Using these specifications, the hardware generator converts the final architecture into a functional and synthesizable design that can efficiently run the analytics algorithm.

### 3.6.2 Compiler

The compiler schedules, maps, and generates the micro-instructions for both ACs and AUs for each sub-node in the $h$DFG. For scheduling and mapping a node, the compiler keeps track of the sequence of scheduled nodes assigned to each AC and AU on a per-cycle basis. For each node which is "ready", i.e., all its predecessors have been scheduled, the compiler tries to place that operation with the goal to improve throughput. Elementary and non-linear operation nodes are spread across as many AUs as required by the dimensionality of the operation. As these operations are completely parallel and do not have any data dependencies within a node, they can be dispersed. For instance, an element-wise vector-vector multiplication, where each vector contains 16 scalar values will be scheduled across two ACs (8 AUs per ACs). Group operations exhibit data dependencies, hence, they are mapped to minimize the communication cost. After all the sub-nodes are mapped, the compiler generates the AC and AU micro-instructions.

The FPGA design, its schedule, operation map, and instructions are then stored in the RDBMS catalog. These components are executed when the query calls for the corresponding UDF.

**Table 3.5:** Descriptions of datasets and machine learning models used for evaluation. Shaded rows are synthetic datasets.

| Workloads | Machine Learning Algorithm | Model Topology | Training Data | | |
|---|---|---|---|---|---|
| | | | # of Tuples | # 32KB Pages | Size (MB) |
| Remote Sensing | Logistic Regression, SVM | 54 | 581,102 | 4,924 | 154 |
| WLAN | Logistic Regression | 520 | 19,937 | 1,330 | 42 |
| Netflix | Low Rank Matrix Factorization | 6040, 3952, 10 | 6,040 | 3,068 | 96 |
| Patient | Linear Regression | 384 | 53,500 | 1,941 | 61 |
| Blog Feedback | Linear Regression | 280 | 52,397 | 2,675 | 84 |
| S\N Logistic | Logistic Regression | 2,000 | 387,944 | 96,986 | 3,031 |
| S\N SVM | SVM | 1,740 | 678,392 | 169,598 | 5,300 |
| S\N LRMF | Low Rank Matrix Factorization | 19880, 19880, 10 | 19,880 | 50,784 | 1,587 |
| S\N Linear | Linear Regression | 8,000 | 130,503 | 130,503 | 4,078 |
| S\E Logistic | Logistic Regression | 6,033 | 1,044,024 | 809,339 | 25,292 |
| S\E SVM | SVM | 7,129 | 1,356,784 | 1,242,871 | 38,840 |
| S\E LRMF | Low Rank Matrix Factorization | 28002, 45064, 10 | 45,064 | 162,146 | 5,067 |
| S\E Linear | Linear Regression | 8000 | 1,000,000 | 1,027,961 | 32,124 |

## 3.7 Evaluation

We prototype DAnA by integrating it with PostgreSQL and compare the end-to-end runtime performance of DAnA generated accelerators with a popular scalable in-database advanced analytics library, Apache MADlib [34, 35], for both PostgreSQL and Greenplum RDBMSs. We compare the end-to-end runtime performance of these three systems. Next, we investigate the impact of *Striders* on the overall runtime of DAnA and how accelerator performance varies with the system parameters. Such parameters include the buffer page-size, number of Greenplum segments, multi-threading on the hardware accelerator, and bandwidth and compute capability of the target FPGA. We also aim to understand the overheads of performing analytics within RDBMS, thus compare MADlib+PostgreSQL with software libraries optimized to perform analytics outside the database. Furthermore, to delineate the overhead of reconfigurable architecture, we compare our FPGA designs with custom hard coded hardware designs targeting a single or fixed set of machine learning algorithms.

**Table 3.6:** Xilinx Virtex UltraScale+ VU9P FPGA specifications.

| FPGA Capacity | | Frequency | BRAM Size | # DSPs |
|---|---|---|---|---|
| 1,182 K LUTS | 2,364 K Flip-Flops | 150 MHz | 44 MB | 6,840 |

**Datasets and workloads.** Table 3.5 lists the datasets and machine learning models used to evaluate DAnA. These workloads cover a diverse range of machine learning algorithms, – Logistic Regression (Logistic), Support Vector Machines (SVM), Low Rank Matrix Factorization (LRMF), and Linear Regression (Linear). Remote Sensing, WLAN, Patient, and Blog Feedback are publicly available datasets, obtained from the UCI repository [44]. Remote Sensing is a classification dataset used by both logistic regression and support vector machine algorithms. Netflix is a movie recommendation dataset for LRMF algorithm. The model topology, number of tuples, and number of uncompressed 32 KB pages that fit the entire training dataset are also listed in the table. Publicly available datasets fit entirely in the buffer pool, hence impose low I/O overheads. To evaluate the performance of out-of-memory workloads, we generate eight synthetic datasets. Synthetic Nominal (S\N) and Synthetic Extensive (S\E) datasets are used to evaluate performance with the increasing sizes of datasets and model topologies. Finally, Table 3.7 provides absolute runtimes for all workloads across our three systems.

**Experimental setup.** We use the Xilinx Virtex UltraScale+ VU9P as the FPGA platform for DAnA and synthesize the hardware at 150 MHz using Vivado 2018.2. Specifications of the FPGA board are provided in Table 3.6. DAnA accelerators . The baseline experiments for MADlib were performed on a machine with four Intel i7-6700 cores at 3.40GHz running Ubuntu 16.04 xLTS with kernel 4.8.0-41, 32GB memory, a 256GB Solid State Drive storage. We run each workload with MADlib v1.12 on PostgreSQL v9.6.1 and Greenplum v5.1.0 to measure single- and multi-threaded performance, respectively.

**Default setup.** Our default setup uses a 32 KB buffer page size and 8 GB buffer pool size across all the systems. As DAnA operates with uncompressed pages to avoid on-chip

**Table 3.7:** Absolute runtimes across all systems.

| Workloads | MADlib+PostgreSQL | MADlib+Greenplum | DAnA+PostgreSQL |
|---|---|---|---|
| Remote Sensing LR | 3s 600ms | 1s 100ms | 0s 100ms |
| WLAN | 14s 0ms | 14s 0ms | 0s 610ms |
| Remote Sensing SVM | 1s 700ms | 0s 600ms | 0s 90ms |
| Netflix | 62s 300ms | 69s 200ms | 7s 890ms |
| Patient | 2s 800ms | 0s 900ms | 1s 180ms |
| Blog Feedback | 1s 600ms | 0s 500ms | 0s 340ms |
| S/N Logistic | 54m 52s | 49m 53s | 2m 11s |
| S/N SVM | 56m 26s | 12m 50s | 4m 4s |
| S/N LRMF | 0m 23s | 0m 3s | 0m 2s |
| S/N Linear | 29m 7s | 24m 16s | 5m 35s |
| S/E Logistic | 66h 45m 0s | 8h 30m 0s | 0h 11m 24s |
| S/E SVM | 0h 6m 0s | 0h 5m 24s | 0h 1m 12s |
| S/E LRMF | 0h 54m 36s | 0h 26m 24s | 0h 39m 0s |
| S/E Linear | 6h 36m 36s | 5h 22m 12s | 0h 16m 48s |

decompression overheads, 32 KB pages are used as a default to fit at least 1 tuple per page for all the datasets. To understand the performance sensitivity by varying the page size on PostgreSQL and Greenplum, we measured end-to-end runtimes for 8, 16, and 32 KB page sizes. We found that page size had no significant impact on the runtimes. Additionally, we did a sweep for 4, 8, and 16 segments for Greenplum. We observed the most benefits with 8 segments, making it our default choice. Results are obtained for both warm cache and cold cache settings to better interpret the impact of I/O on the overall runtime. In the case of a warm cache, before query execution, training data tables for the publicly available dataset reside in the buffer pool, whereas only a part of the synthetic datasets are contained in the buffer pool. For the cold cache setting, before execution, no training data tables reside in the buffer pool.

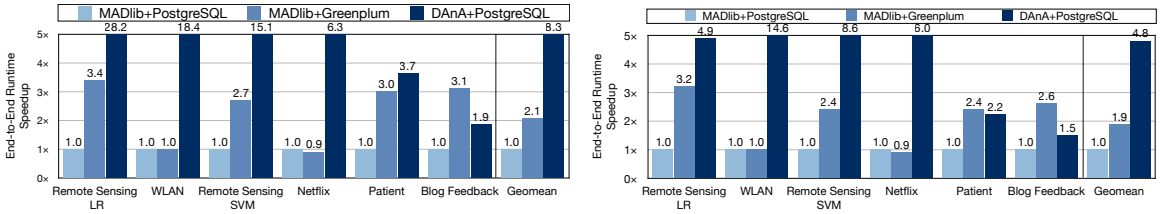
### 3.7.1 End-to-End Performance

**Publicly available datasets.** Figures 3.8a and 3.8b illustrate end-to-end performance of MADlib+PostgreSQL, Greenplum+MADlib, and DAnA, for warm and cold cache. The x-

axis represents the individual workloads and y-axis the speedup. The last bar provides the geometric mean (geomean) across all workloads. On average, DAnA provides $8.3\times$ and $4.8\times$ end-to-end speedup over PostgreSQL and $4.0\times$ and $2.5\times$ speedup over 8-segment Greenplum for publicly available datasets in warm and cold cache setting, respectively. The benefits diminish for cold cache as the I/O time adds to the runtime and cannot be parallelized. The overall runtime of the benchmarks reduces from 14 to 1.3 seconds with DAnA in contrast to MADlib+PostgreSQL.

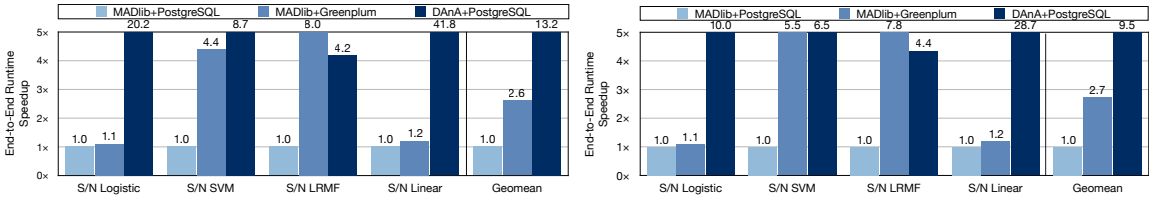
The maximum speedup is obtained by Remote Sensing LR, $28.2\times$ with warm cache and $14.6\times$ with cold cache. This workload runs logistic regression algorithm to perform non-linear transformations to categorize data in different classes and offers copious amounts of parallelism for exploitation by DAnA's accelerator. In contrast, Blog Feedback sees the smallest speedup of $1.9\times$ (warm cache) and $1.5\times$ (cold cache) due to the high CPU vectorization potential of the linear regression algorithm.

**Synthetic nominal and extensive datasets.** Figures 3.9 and 3.10 depict end-to-end performance comparison for synthetic nominal and extensive datasets across our three systems. Across S/Ndatasets, shown in Figure 3.9, DAnA achieves an average speedup of $13.2\times$ in warm cache and $9.5\times$ in cold cache setting. In comparison to 8-segment Greenplum, for S/Ndatasets, DAnA observes a gain of $5.0\times$ for warm cache and $3.5\times$ for cold cache. The average speedup as shown in Figure 3.10, across S/Edatasets in comparison to MADlib+PostgreSQL are $12.9\times$ for warm cache and $11.9\times$ for cold cache. These speedups reduce to $5.9\times$ (warm cache) and $7.0\times$ (cold cache) when compared against 8-segment MADlib+Greenplum. Higher benefits of acceleration are observed with larger datasets as DAnA accelerators are exposed to more opportunities for parallelization, which enables the accelerator to hide the overheads such as data transfer across platforms, on-chip data alignment, and setting up the execution pipeline. These results show the efficacy of the multi-threading employed by DAnA's execution engine in comparison to the scale-out



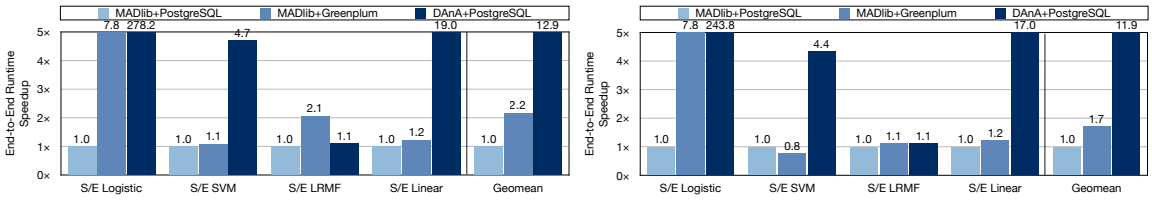

**(a)** End-to-End Performance for Warm Cache **(b)** End-to-End Performance for Cold Cache

**Figure 3.8:** End-to-end runtime performance comparison for publicly available datasets with MADlib+PostgreSQL as baseline.




**(a)** End-to-End Performance for Warm Cache **(b)** End-to-End Performance for Cold Cache

**Figure 3.9:** End-to-end runtime performance comparison for synthetic nominal datasets with MADlib+PostgreSQL as baseline.




**(a)** End-to-End Performance for Warm Cache **(b)** End-to-End Performance for Cold Cache

**Figure 3.10:** End-to-end runtime performance comparison for synthetic extensive datasets with MADlib+PostgreSQL as baseline.

Greenplum engine. The total runtime for S/N Logistic and S/E Logistic reduces from 55 minutes to 2 minutes and 66 hrs to 12 minutes, respectively. These workloads achieve high reduction in total runtimes due to their high skew towards compute time, which DAnA's execution engine is specialized in handling. For S/N SVM, DAnA is only able to reduce the runtime by 20 seconds. This can be attributed to the high I/O time incurred by the benchmark in comparison to its compute time, thus, the accelerator frequently stalls for the buffer pool page replacements to complete. Nevertheless, for S/E SVM, DAnA still reduces the absolute runtime from 55 to 39 minutes.

**Evaluating *Striders*.** A crucial part of DAnA's accelerators is their direct integration with the buffer pool via *Striders*. To evaluate the effectiveness of *Striders*, we simulate an alternate design where DAnA's execution engines are fed by the CPU. In this alternative, the CPU transforms the training tuples and sends them to the execution engines. Figure 3.11 compares the end-to-end runtime of DAnA with and without *Striders* using warm cache MADlib+PostgreSQL as baseline. DAnA with and without *Striders* achieve, on average, $10.7\times$ and $2.3\times$ speedup in comparison to the baseline. Even though raw application hardware acceleration has its benefits, integrating *Striders* to directly interface with the database engine amplifies those performance benefits by $4.6\times$. The *Striders* bypass the bottlenecks in the memory subsystem of CPUs and provide an on-chip opportunity to intersperse the tasks of the access and execution engines. The above evaluation demonstrates the effectiveness of DAnA and *Striders* in integrating with PostgreSQL.

### 3.7.2 Performance Sensitivity

**Multi-threading in Greenplum.** As shown in Figure 3.12, for publicly available datasets, the default configuration of 8-segment Greenplum provides $2.1\times$ (warm cache) and $1.9\times$ (cold cache) higher speedups than its PostgreSQL counterpart. The 8-segment Greenplum performs the best amongst all options and performance does not scale as the

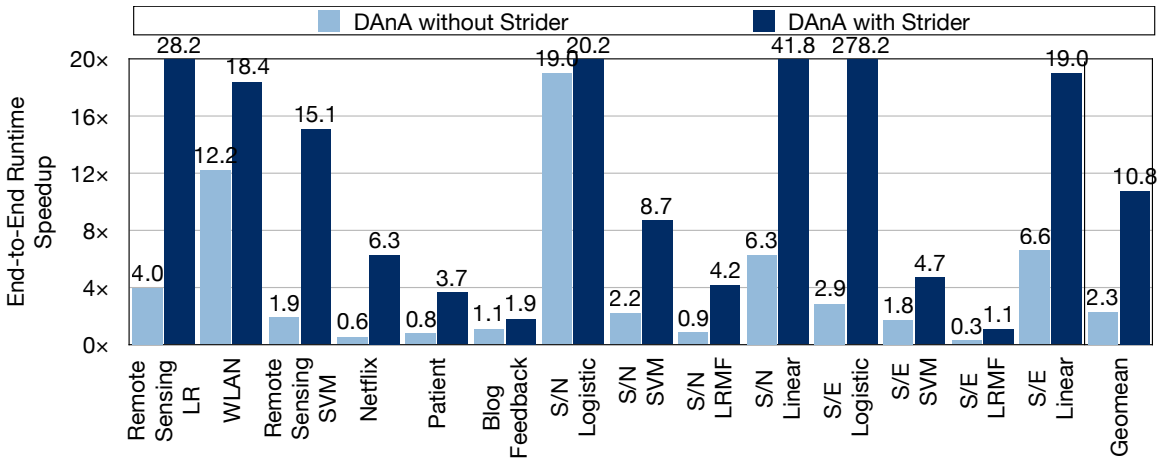



**Figure 3.11:** Comparison of DAnA with and without *Striders* with PostgreSQL +MADlib as the baseline.

segments increase.

**Performance Sensitivity to FPGA resources.** Two main resource constraints on the FPGA are its compute capability and bandwidth. DAnA configures the template architecture in accordance to the algorithmic parameters and FPGA constraints. To maximize compute resource utilization, DAnA supports a multi-threaded execution engine, where each thread runs a version of the update rule. We perform an analysis for varying number of threads on the final accelerator by changing the merge coefficient. A merge coefficient of 2 implies a maximum of two threads. However, a large merge coefficient, such as 2048, does not warrant 2048 threads, as the FPGA may not have enough resources. In UltraScale+ FPGA, maximum 1024 compute units can be instantiated.

Figure 3.13 shows performance sensitivity with increasing compute utilization of FPGA for different workloads. Each plot shows DAnA's accelerator runtime (access engine + execution engine) in comparison to a single-thread. The sensitivity towards compute resources is a function of algorithm type, model width, and # of epochs. Thus, each workload fares differently with varying compute resources. Workloads such as Remote Sensing LR and Remote Sensing SVM have a narrow model size, thus, increasing the number of threads increases performance till they reach peak compute utilization. On the other hand, LRMF

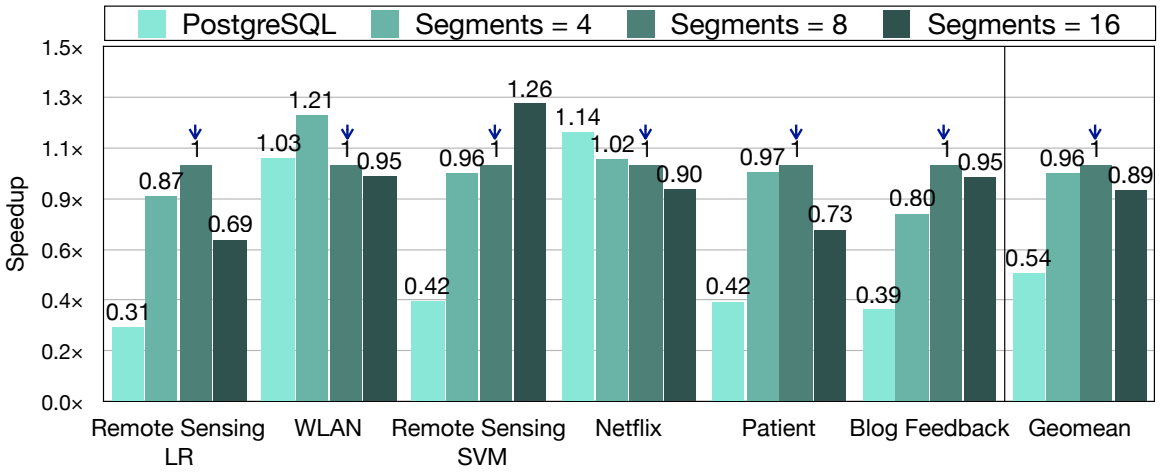



**Figure 3.12:** Greenplum performance with varying segments.

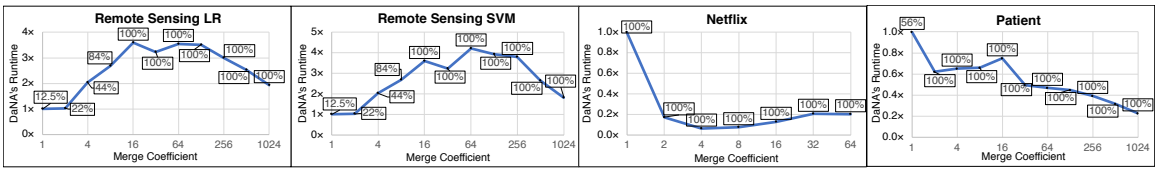



**Figure 3.13:** Runtime performance of DAnA with increasing # of threads compared with single-thread as baseline.

algorithm workloads do not experience a higher performance with increasing number of threads. This can be attributed to the copious amounts of parallelism available in a single instance of the update rule. Thus, increasing the number of threads reduces the ACs allocated to a single thread, whereas, merging across multiple different threads incurs an overhead. One of the challenges tackled by the compiler is to allocate the on-chip resources by striking a balance between the single-thread performance and multi-thread parallelism.

Figure 3.14 illustrates the impact of FPGA bandwidth (in comparison to baseline bandwidth) on the speedup of DAnA accelerators. The results show that as the size of the benchmark increases, except the ones that run LRMF algorithm, the workloads become bandwidth bound. The workloads S/N LRMF and S/E LRMF are compute heavy, thus, bandwidth increase does not have a significant impact on the accelerator runtime.

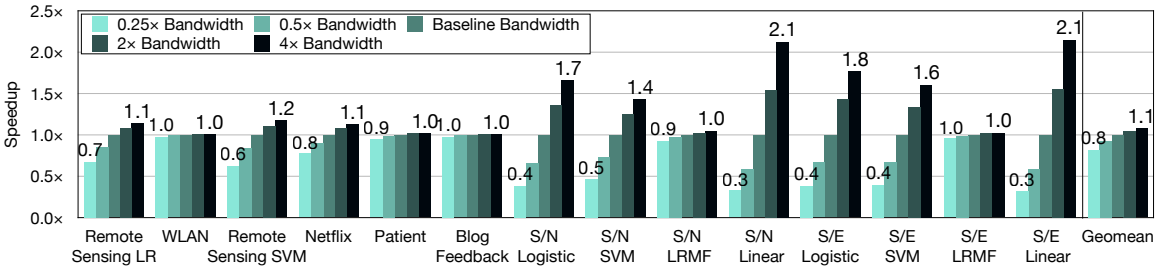



**Figure 3.14:** Comparison of FPGA time with varying bandwidth.

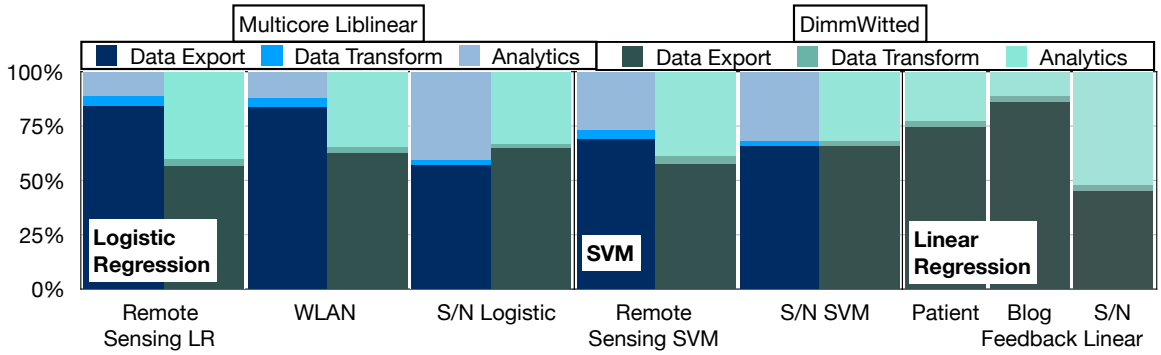
### 3.7.3 Comparison to Custom Designs

Potential alternatives to DAnA are: (1) custom software libraries [95, 51, 96] that run multi-core advanced analytics and (2) algorithm specific FPGA implementations [97, 98, 99]. For these alternatives, if training data is stored in the database, there is an overhead to extract, transform, and supply the data in accordance to each of their requirements. We compare the performance of these alternatives with MADlib+PostgreSQL and DAnA.

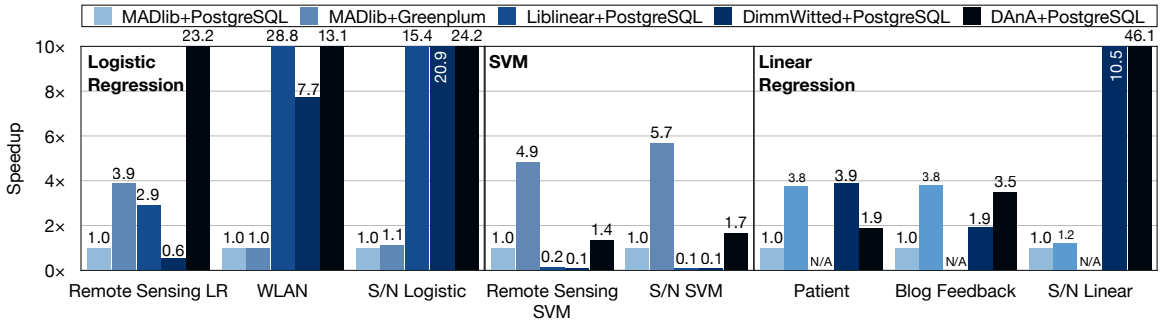
**Optimized software libraries.** We compare C++-optimized libraries DimmWitted and Liblinear-Multicore classification with MADlib+PostgreSQL, Greenplum+MADlib, and DAnA accelerators. Liblinear supports Logistic Regression and SVM, and DimmWitted supports SVM, Logistic Regression, Linear Regression, Linear Programming, Quadratic Programming, Gibbs Sampling, and Neural Networks. Logistic Regression, SVM, and Linear Regression (only DimmWitted), overlap with our benchmarks, thus, we compare multi-core versions (2, 4, 8, 16 threads) of these libraries and use the minimum runtime. We maintain the same hyper-parameters, such as tolerance, and choice of optimizer to compare runtime of 1 epoch across all the systems. We separately compare the compute time and end-to-end runtime (data extraction from PostgreSQL + data transformation + compute), as well as provide a runtime breakdown. Figure 3.15a illustrates the breakdown of Liblinear and DimmWitted into the different phases that comprise the end-to-end runtime. Data exporting and reformatting for these external specialized machine learning tools is an overhead specific to performing analytics outside RDBMS. Results suggest that DAnA is

uniformly faster, as it (1) does not export the data from the database, (2) employs *Striders* in the FPGA to walk through the data, and (3) accelerates the machine learning computation with an FPGA. However, different software solutions exhibit different trends, as elaborated below.

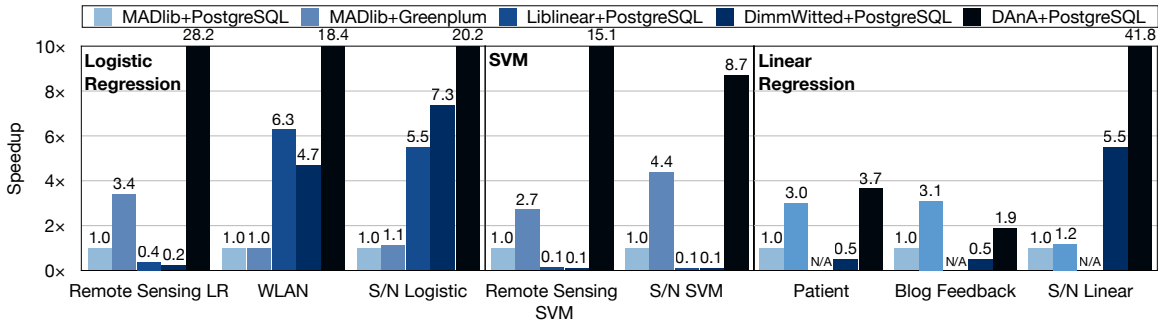
- **Logistic regression:** As Figure 3.15b shows, Liblinear and DimmWitted provide 3.8× and 1.8× speedup over MADlib+PostgreSQL for logistic regression-based workloads in terms of compute time. With respect to end-to-end runtime compared to MADlib+PostgreSQL (Figure 3.15c), the benefits from Liblinear reduce to 2.4× and increase for DimmWitted to 2.1×. On the other hand, DAnA outperforms Liblinear by 2.2× and DimmWitted by 4.7× in terms of compute time. For overall runtime, DAnA is 9.1× faster than Liblinear and 10.4× faster than DimmWitted. The compute time of Remote sensing LR benchmark receives the least benefit from LibLinear and DimmWitted and exhibits a slowdown in end-to-end runtime. This can be attributed to the small model size, which, despite a large dataset, does not provide enough parallelism that can be exploited by these libraries. Specifically sparse datasets, such as WLAN, are handled more efficiently by these libraries.
- **SVM:** As shown in Figure 3.15b, that compares compute time of SVM-based workloads, Liblinear and DimmWitted are 18.1× and 22.3× slower than MADlib+PostgreSQL, respectively. For end-to-end runtime (Figure 3.15c), the slowdown is reduced to 14.6× for Liblinear and 15.9× for DimmWitted, due to the complex interplay between data accesses and UDF execution of MADlib+PostgreSQL. In comparison, DAnA outperforms Liblinear by 30.7× and DimmWitted by 37.7× in terms of compute time. For overall runtime, DAnA is 127× and 138.3× faster than Liblinear and DimmWitted, respectively.
- **Linear regression:** For linear regression-based workloads, DimmWitted is 4.3× faster than MADlib+PostgreSQL in terms of compute time. For end-to-end runtime compared to MADlib+PostgreSQL (Figure 3.15c), the speedup of DimmWitted is reduced to 12%. DAnA outperforms DimmWitted by 1.6× and 6.0× in terms of compute and overall time,




**(a)** Runtime breakdown for Liblinear and DimmWitted




**(b)** Compute Time Comparison




**(c)** End-to-End Runtime Comparison

**Figure 3.15:** Comparison to external software libraries.

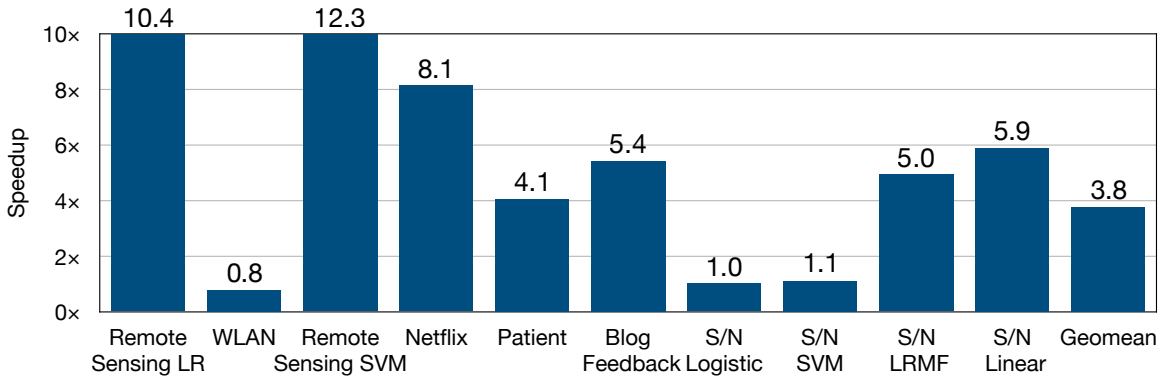



**Figure 3.16:** Performance comparison of DAnA over TABLA.

respectively.

**Specific FPGA implementations.** We compare hand-optimized FPGA designs created specifically for one algorithm with our reconfigurable architecture. DAnA's execution engine performance is on par with Parallel SVM [97], is 44% slower than Heterogeneous SVM [98], and is 1.47× faster than Falcon Logistic Regression [99]. In addition to the speedup, we compare Giga Ops Per Second (GOPS), to measure the numerical compute performance of these architectures. In terms of GOPS, DAnA performs, on average, 16% less operations than these hand-coded designs. In addition to providing comparable performance, DAnA relieves the data scientist of the arduous task of hardware design and testing whilst integrating seamlessly within the database engine. Whereas, for these custom designs, designer requires hardware design expertise and long verification cycles to write ≈15000 lines of Verilog code.

**Comparison with TABLA.** We compare DAnA with TABLA [12], an open-source framework [100] that generates optimized FPGA implementations for a wide variety of analytics algorithms. We modify the templates for UltraScale+ and perform design space exploration to present the best case results with TABLA. Figure 3.16 shows that DAnA generated accelerators perform 4.7× faster than TABLA accelerators. DAnA's benefits can be attributed to the: (1) interleaving of *Striders* in the access engine with the execution engine to mitigate

the overheads of data transformation and (2) the multi-threading capability of the execution engines to exploit parallelism between different instances of the update rule. TABLA on the other hand, offers only single threaded acceleration.

## 3.8 Related Work

**Hardware acceleration for data management.** Accelerating database operations is a popular research direction that connects modern acceleration platforms and enterprise in-database analytics as shown in Figure 3.1. These prior FPGA-based solutions aim to accelerate DBMS operations (some portion of the query) [84, 85, 89, 90, 82], such as join and hash. LINQits [82] accelerates database queries but does not focus on machine learning. Centaur [84] dynamically decides which particular operators in a MonetDB [101] query plan can be executed on FPGA and creates a pipeline between FPGA and CPU. Another work [90] uses FPGAs to provide a robust hashing mechanism to accelerate data partitioning in database engines. In the GPU realm, HippogriffDB [88] aims to balance the I/O and GPU bandwidth by compressing the data that is transferred to GPU. Support for in-database advanced analytics for FPGAs in tandem with *Striders* set this work apart from the aforementioned literature, which does not focus on providing components that integrate FPGAs within an RDBMS engine and machine learning.

**Hardware acceleration for advanced analytics.** Both research and industry have recently focused on hardware acceleration for machine learning [57, 12, 70, 91] especially deep neural networks [102, 103, 104, 105, 106] connecting two of the vertices in Figure 3.1 traid. These works either only focus on a fixed set of algorithms or do not offer the reconfigurability of the architecture. Among these, several works [93, 107, 86] provide frameworks to automatically generate hardware accelerators for stochastic gradient descent. However, none of these works provide hardware structures or software components that embed FPGAs within the RDBMS engine. DAnA's Python DSL builds upon

the mathematical language in the prior work [12, 86]. However, the integration with both conventional (Python) and data access (SQL) languages provides a significant extension by enabling support for UDFs which include general iterative update rules, merge functions, and convergence functions.

**In-Database advanced analytics.** Recent work at the intersection of databases and machine learning are extensively trying to facilitate efficient in-database analytics and have built frameworks and systems to realize such an integration [108, 109, 110, 19, 34, 35, 111, 112, 37, 113, 114] (see [115] for a survey of various methods and systems). DAnA takes a step forward and exposes FPGA acceleration for in-Database analytics by providing a specialized component, *Strider*, that directly interfaces with the database to alleviate some of the shortcomings of the traditional Von-Neumann architecture in general purpose compute systems. Past work in Bismarck [19] provides a unified architecture for in-database analytics, facilitating UDFs as an interface for the analyst to describe their desired analytics models. However, unlike DAnA, Bismarck lacks the hardware acceleration backend and support for general iterative optimization algorithms.

## 3.9 Conclusion

This paper aims to bridge the power of well-established and extensively researched means of structuring, defining, protecting, and accessing data, i.e., RDBMS with FPGA accelerators for compute-intensive advanced data analytics. DAnA provides the initial coalescence between these paradigms and empowers data scientists with no knowledge of hardware design, to use accelerators within their current in-database analytics procedures.

# CHAPTER 4
# ENABLING METHODICAL AND CONTROLLED APPROXIMATION FOR HIGH PERFORMANCE HARDWARE DESIGNS

## 4.1 Summary

Relaxing the traditional abstraction of "near-perfect" accuracy in hardware can yield significant gains in efficiency, area, and performance. To exploit this opportunity, there is a need for design abstractions and synthesis tools that can systematically incorporate approximation in a hardware design. We define Axilog, a set of backward compatible language extensions for Verilog, that provide the necessary syntax and semantics for approximate hardware design and reuse. Axilog enables designers to safely relax the accuracy requirements in the design, while keeping the critical parts strictly precise. Axilog is coupled with a Safety Inference Analysis that automatically infers the safe-to-approximate gates and connections from the annotations. The analysis provides formal guarantees that the safe-to-approximate parts of the design are in strict accordance to the designer's intentions. We evaluate Axilog using a diverse set of benchmarks that gain 1.54× average energy savings and 1.82× average area reduction with 10% output quality loss. The results show that the intuitive nature of the language extensions coupled with the automated analysis enables safe approximation of designs even with thousands of lines of code.

Axilog opens new avenues for hardware designers to express a design which can generate approximate results. , Such a design is most commonly deployed as an approximate accelerator (an ad-hoc addition to the CPU), best suitable to run a compute intensive but approximation amenable part of an application. Often such kernels are repeated multiple times to amortize the cost of data transfer back and forth to the accelerator; conventionally, this accelerator will execute every invocation of this frequently executed code region

without considering the final application quality degradation. However, once such a design is deployed, its outputs will always be approximate in accordance to the inputs, and the designer will have no control over the degree of approximation. Instead, there is a vast decision space in which each invocation can either be delegated to the accelerator—improving performance and efficiency–or run on the precise core—maintaining quality. Therefore, we devise a mechanism called MITHRA, which is a co-designed hardware-software solution, that navigates these tradeoffs to deliver high performance and efficiency while lowering the final quality loss.

MITHRA seeks to identify whether each individual accelerator invocation will lead to an undesirable quality loss and, if so, directs the processor to run the original precise code. This identification is cast as a binary classification task that requires a cohesive hardware-software co-design. The hardware component performs the classification at runtime and exposes a knob to the software mechanism to control quality tradeoffs. The software tunes this knob by solving a *statistical optimization problem* that maximizes benefits from approximation while providing statistical guarantees that final application quality level will be met with high confidence. The software tunes this knob to train our two distinct hardware classifiers – one table-based and one neural network based. We evaluate an integrated system consisting of a CPU and an approximate accelerator augmented with these two instances of MITHRA. To understand the efficacy of these mechanisms, we compare with an ideal, but infeasible design, *the oracle*. Results show that, with 95% confidence the table-based design can restrict the final output quality loss to 5% for 90% of unseen input sets while providing 2.5× speedup and 2.6× energy efficiency. The neural design shows similar speedup however, improves the efficiency by 13%. Compared to the table-based design, the oracle improves speedup by 26% and efficiency by 36%. Results show that MITHRA performs within a close range of the oracle and can effectively navigate the quality tradeoffs in approximate acceleration.

In this chapter, we first delve into the language design and constructs of Axilog, and

then discuss in detail the entire MITHRA framework for quality control.

## 4.2 Abstractions for Approximate Hardware Design and Reuse

Several works have shown significant benefits with approximation at the circuit level [116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130]. While these techniques allow circuit-level approximation, there is a lack of design abstractions that enable designers to methodically control which parts of a circuit can be approximated while keeping the critical parts precise. Thus, there is a need for *approximate hardware description languages* enabling systematic synthesis of approximate hardware. We introduce Axilog–a set of concise, intuitive, and high-level annotations–that provides the necessary syntax and semantics for approximate hardware design and reuse in Verilog.

Axilog enables designers to delineate which parts of a hardware system or circuit are critical and cannot be approximated. A key factor in our language formalism is to abstract away the details of approximation while maintaining the designer's complete oversight in deciding which circuit elements can be synthesized approximately and which circuit elements are critical and cannot be approximated. Axilog also supports reusability across modules by providing a set of specific reuse annotations. In general, hardware system implementation relies on modular design practices where the engineers build libraries of modules and reuse them across complex hardware systems. Section 4.2.1 elaborates on the Axilog annotations for approximate hardware design and reuse. These annotations are coupled with Safety Inference Analysis that automatically infers which circuit elements are safe-to-approximate (Section 4.2.2) with respect to the designer's annotations. The Safety Inference Analysis formally guarantees that approximation will only affect the circuit elements that the designer intended to approximate. Section 4.2.2 details this analysis. Axilog and safety analysis support approximate synthesis, however, they are completely independent of the synthesis process. To evaluate Axilog, we devised two synthesis processes (Section 4.2.3). The first synthesis flow focuses on current technology nodes and leverages

**Table 4.1:** Summary of Axilog's language syntax.

| Phase | Annotation | Arg | Description |
|---|---|---|---|
| Design | relax | wire, reg, output, inout | Declare an argument as safe-to-approximate. Design element that affect the argument are safe to approximate. |
| | relax_local | | Similar to relax but the approximation does not cross module boundaries. |
| | restrict | | Any design element that affects the argument is made precise unless explicitly relaxed. |
| | restrict_global | | All the design elements affecting the argument are precise. |
| Reuse | approximate | output, inout | Indicates the output carries relaxed sematics. |
| | critical | input | Indicates the input is critical and approximate elements cannot drive it. |
| | bridge | wire, reg | Allow connecting an approximate element to a critical input. |

commercial tools. This synthesis process applies approximation by relaxing the timing constraints of the safe-to-approximate sub-circuits. The results show that this synthesis flow provides, on average, 1.54× energy savings and 1.82× area reduction by allowing a 10% quality loss. The second synthesis flow aims to study the potential of approximate synthesis by using probabilistic gate model for future technology nodes. Since the characteristics of gates for future technologies are unknown we assume that the probability of error for a gate is an inverse function of its size. The results show that the second synthesis flow provides, on average, 2.5× energy and 2.2× PCMOS area reduction (defined in Section 4.2.3.2). Axilog yields these significant benefits while only requiring between 2 to 12 annotations even with complex designs containing up to 22,407 lines of code. These results confirm the effectiveness of Axilog in incorporating approximation in the hardware design cycle.

### 4.2.1 Approximate Hardware Design

Our principle objectives for approximate hardware design with Axilog are (1) to craft a small number of Verilog annotations that provide designers with complete oversight over the approximation process; (2) to minimize the number of manual annotations while relying on Safety Inference Analysis (Section 4.2.2) to automatically infer the designer's intent for approximation. This relieves the designer from the details of the approximate synthesis process; (3) to support the reuse of Axilog modules across different designs without the

need for reimplementation. Furthermore, Axilog is a backward-compatible extension of Verilog. That is, an Axilog code with no annotations is a normal Verilog code. To this end, Axilog provides two sets of language extensions, one set for the design (Section 4.2.1.1) and the other for the reuse of hardware modules (Section 4.2.1.2). Table 4.1 summarizes the syntax for the design and reuse annotations. The annotations for design dictate which operations and connections are safe-to-approximate in the module. Henceforth, for brevity, we refer to operations and connections as design elements. The annotations for reuse enable designers to use the annotated approximate modules across various designs without any reimplementation. We provide detailed examples to illustrate how designers are able to appropriately relax or restrict the approximation in hardware modules. In the examples, we use **background shading** to highlight the safe-to-approximate elements inferred by the analysis.

#### *4.2.1.1 Design Annotations*

**Relaxing accuracy requirements.** By default, all design elements are precise. The designer can use the relax(arg) statement to *implicitly* approximate a subset of these elements. The variable arg is either a wire, reg, output, or inout. Design elements that *exclusively* affect signals designated by the relax annotation are safe to approximate. The use of relax is illustrated using the following example.

```
module full_adder(a, b, c_in, c_out, s);
    input a, b, c_in; output c_out;
    approximate output s;
    assign s = a ^ b ^ c_in;
    assign c_out = a & b + b & c_in + a & c_in;
    relax(s);
endmodule
```

In this full_adder example, the relax(s) statement implies that the analysis can automatically approximate the XOR operations. The unannotated c_out signal and the logic generating it is not approximated. Furthermore, since s will carry relaxed semantics, its corresponding output is marked with the approximate annotation that is necessary for reusing

modules (discussed in Section 4.2.1.2). With these annotations and the automated analysis, the designer does not need to *individually* declare the inputs (a, b, c_in) or any of the XOR (^) operations as approximate. Thus, while designing approximate hardware modules, this abstraction significantly reduces the burden on the designer to understand and analyze complex data flows within the circuit.

**Scope of approximation.** The scope of the relax annotation crosses the boundaries of instantiated modules as shown by the code on the left side. The relax(x) annotation in the nand_gate module implies that the AND(&) operation in the and_gate module is safe-to-approximate. In some cases, the designer might not prefer the approximation to cross the scope of the instantiated modules. Axilog provides the relax_local annotation that does not cross module boundaries.

```
module and_gate(n,a,b);
   input a, b; output n;
   assign n = a & b;
endmodule
module nand_gate(x, a, b);
   input a, b;
   approximate output x;
   wire w0;
   and_gate a1(w0, a, b);
   assign x = ~ w0;
   relax(x);
endmodule
```

```
module and_gate(n,a,b);
   input a,b; output n;
   assign n = a & b;
endmodule
module nand_gate(x, a, b);
   input a, b;
   approximate output x;
   wire w0;
   and_gate a1(w0, a, b);
   assign x = ~ w0;
   relax_local(x);
endmodule
```

The code on the right side shows that the relax_local annotation does not affect the semantics of the instantiated and_gate module, a1. However the NOT (~) operation which shares the scope of the relax_local annotation is safe-to-approximate. The scope of approximation for both relax and relax_local is the module in which they are declared.

**Restricting approximation.** In some cases, the designer might want to *explicitly* restrict approximation in certain parts of the design. Axilog provides the restrict(arg) annotation that

ensures that any design element affecting the annotated argument (arg) is precise, *unless* a preceding relax or relax_local annotation has made the driving elements safe-to-approximate. The restrict annotation crosses the boundary of instantiated modules.

**Restricting approximation globally.** There might be cases where the designer intends to override preeceding relax annotations. For instance, the designer might intend to keep certain design elements that are used to drive critical signals such as the control signals for a state machine, write enable of registers, address lines of a memory module, or even clock and reset. To ensure the precision of these signals Axilog provides the restrict_global annotation that has precedence over relax and relax_local. The restrict_global(arg) penetrates through module boundaries and ensures that any design element that affects arg is not approximated.

#### *4.2.1.2 Reuse Annotations*

Our principle idea behind these language abstractions is to maximize the reusability of the approximate modules across designs that may have different accuracy requirements. This section describes the abstractions that are necessary for reusing approximate modules.

**Outputs carrying approximate semantics.** As mentioned before, designers can use annotations to selectively approximate design elements in a module. The reusing designer needs to be aware of the accuracy semantics of the input/output ports without delving into the details of the module. To enable the reusing designer to view the port semantics, Axilog requires that all output ports that might be influenced by approximation to be marked as approximate. Below, the code snippets illustrate the necessity of the approximate annotation.

```
module and_gate(n,a,b);
    input a,b;
    approximate output n;
    assign n = a & b;
    relax n;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
endmodule
```

```
module and_gate(n,a,b);
    input a, b;
    output n;
    assign n = a & b;
endmodule
module nand_gate(x, a, b);
    input a, b;
    approximate output x;
    wire w0;
    and_gate a1(w0, a, b);
    assign x = ~ w0;
    relax (x);
endmodule
```

On the left side, output n carries relaxed semantics due to the relax annotation and is therefore declared as an approximate output. Consequently, the a1 instance in the nand_gate module will cause its x output to be relaxed. Therefore, x is marked as an approximate output. On the right side, the x output is explicitly relaxed and x is marked as an approximate output. The and_gate module here does not carry approximate semantics by default. Therefore, the output of the and_gate is not marked as approximate as the approximation is only limited to the a1 instance.

**Critical inputs.** A designer may want to prevent approximation from affecting certain inputs, which are critical to the functionality of the circuit. To mark these input ports, Axilog provides critical annotation. Wires that carry approximate semantics cannot drive the critical inputs without the designer's explicit permission at the time of reuse.

**Bridging approximate wires to critical inputs.** We recognize that there may be cases when the reusing designer entrusts a critical input with an approximate driver. For such situations, Axilog provides an annotation called bridge that shows designer's explicit intent to drive a critical input by an approximate signal.

In summary, the semantics of the relax and restrict annotations provide abstractions for designing approximate hardware modules while enabling Axilog to provide *formal guaran-*

*tees* of safety that approximation will only be restricted to design elements that are specifically selected by the designer. Moreover, the approximate output, critical input, and bridge annotations enable reusability of modules across different designs. In addition to the modularity, the design and reuse annotations altogether enable *approximation polymorphism* implying that the modules with approximate semantics can be used in a precise manner and vice-versa without any reimplementation. These abstractions provide a natural extension to the current practices of hardware design and enable designers to apply approximation with full control without adding substantial overhead to the conventional hardware design and verification cycle.

### 4.2.2 Safety Inference Analysis

After the designer provides annotations, the compiler needs to perform a static analysis to find the approximate and precise design elements in accordance with these annotations. This section presents the *Safety Inference Analysis*, a static analysis that identifies these safe-to-approximate design elements. The design elements are primarily organized according to the structure of the circuit and not necessarily on the order of the statements in the HDL source code. This property is a fundamental property of Verilog that is inherited by Axilog. Thus, we first translate the RTL design to primitive gates, while maintaining the module boundaries. Then, we apply the Safety Inference Analysis after the code is translated to primitive gates and the structure of the circuit is identified. Consequently, the Safety Inference Analysis can apply all the annotations while considering the structure of the circuit. We apply the *Safety Inference Analysis* that is a backward slicing algorithm that starts from the annotated wires and iteratively traverses the circuit to identify which wires must carry precise semantics. Subtracting the set of precise wires from all the wires in the circuit yields the safe-to-approximate set of wires. The gates that immediately drive these safe-to-approximate wires are the ones that the synthesis engine can approximate. Figure 4.1(a) illustrates the procedure that identifies the precise wires.

**Inputs:** $\mathbb{M}$: Set of all the ordered modules within the circuit
$\mathbb{R}$: Queue of all the globally restricted wires
**Output:** $\mathbb{P}$: Set of precise wires

Initialize $\mathbb{P} \leftarrow \emptyset$
**for** each $m_i \in \mathbb{M}$ **do**
$I$: Set of all inputs ports in $m_i$
$A$: Set of all wires annotated as relaxed wires in $m_i$
$LA$: Set of all wires annotated as locally relaxed wires $m_i$
$Sink$: Queue of all explicitly restricted wires in $m_i \cup$ Set of unannotated output ports
$UW$: Set of wires driven by modules that are instantiated within $m_i$

**//Phase1: This loop identifies the $m_i$ module's local precise wires ($w_i$)**
Initialize $N \leftarrow \emptyset$ A set of relaxed wires in each module $m_i$
**while** ($Sink \neq \emptyset$) **do**
$w_i \leftarrow Sink$.dequeue()
**if** ($w_i \notin I$ **and** $w_i \notin (A \cup LA)$) **then**
**if** ($w_i \in UW$) **then**
$N$.append($w_i$)
**else**
$\mathbb{P}$.append($w_i$)
**end if**
$Sink$.enqueue(for all input wires of gate $w_i$ in $m_i$)
**end if**
**end while**

**//Phase 2: Identifying the relaxed wires ($w_j$) that are driven by the $m_j$ submodules; the $m_j$ submodules are the instantiated modules in $m_i$**
**for** ($w_j \in UW$) **do**
**if** ($w_j \notin N$ **and** $w_j$ drives wire $\in A$) **then**
$m_j \leftarrow$ module driving the wire $w_j$
$m_j.A$.append($w_j$)
**end if**
**end for**
**end for**

**//Phase 3: Identifying the precise wires ($w_k$) that are globally restricted**
**while** ($\mathbb{R} \neq \emptyset$) **do**
$w_k \leftarrow \mathbb{R}$.dequeue()
$\mathbb{P}$.append($w_k$)
$\mathbb{R}$.append(input wires of the gate that drive $w_k$)
**end while**

(a) Part of Safety Inference Analysis that identifies precise wires according to the designer's annotations

**Inputs:** $K$: Netlist for the entire circuit
$\Theta$: Set of safe-to-approximate gates
$\Sigma$: Error bound on the approximate output
**Output:** $\Re$: Different gate sizes for safe-to-approximate gates

Initialize $\Re \leftarrow$ Minimum gate size
Initialize $\Psi \leftarrow \emptyset$ {Monte Carlo simulation map}
Initialize $\gamma \leftarrow \emptyset$ {Error propagation map}
Initialize $\Pi \leftarrow \emptyset$ {Primary inputs of the safe-to-approximate circuit}
Initialize $\delta \leftarrow \emptyset$ {Queue for primary inputs of the safe-to-approximate circuit}
Initialize $\Phi \leftarrow \emptyset$ {Primary outputs of the safe-to-approximate circuit}
Initialize $\beta \leftarrow \emptyset$ {Fan-in hash-map}

**//Phase 1: Identifying inputs ($\Pi$) and outputs ($\Phi$) of the safe-to-approximate subset of the circuit. //**
**for** each $m_i \in \Theta$ **do**
**if** fanin_of $m_i \not\subset \Theta$ **then**
$\Pi \leftarrow (\Pi \cup \{m_i\})$
enqueue($\delta$,$m_i$)
**else if** $m_i$ fanout $\not\subset \Theta$ **then**
$\Phi \leftarrow (\Phi \cup \{m_i\})$
**end if**
**end for**

**//Phase 2: Performing Monte Carlo Simulations to calculate probability of 1 or 0 ($\Psi$) at every node**
$\Psi \leftarrow$ monte_carlo_simulation ($\delta$,$K$,$\Theta$,$\Psi$)

**//Calculating the initial error map ($\gamma$) for every output node using boolean error propagation**
$\gamma \leftarrow$ boolean_error_propagation ($\delta$,$K$,$\Theta$,$\Psi$,$\gamma$)

**while** ($\exists\ w_i \in \Phi$ s.t. $\Sigma(w_i) < \gamma(w_i)$) **do**
**//Phase 3: Iteratively calculating the fan-in of every output node using back-propagation and adding the gates to ($\beta$)**
**while** ($\exists\ w_i \in \Phi$ s.t $\Sigma(w_i) < \gamma(w_i)$) **do**
$\beta \leftarrow$ Gates $\in \Theta$ that have a path to $w_i$
$\delta \leftarrow$ Primary inputs $\in \Theta$ that have a path to $w_i$
define $m$ -999 **//Max sensitivity initialized**
**//Phase 4: Calculates the sensitivity of each gate to the output error and permanently resizes the gate with highest sensitivity**
$G \leftarrow \emptyset$
**for** each $y_i \in \beta$ **do**
**if** (sensitivity of $y_i > m$) **then**
$m$ = sensitivity of $y_i$
$G \leftarrow y_i$
**end if**
**end for**
$\Re(G) \leftarrow \Re(G)*2$ **//up-size gate permanently**
$\gamma \leftarrow$ boolean_error_propagation ($\delta$,$K$,$\Theta$,$\Psi$,$\gamma$)
**end while**
**end while**

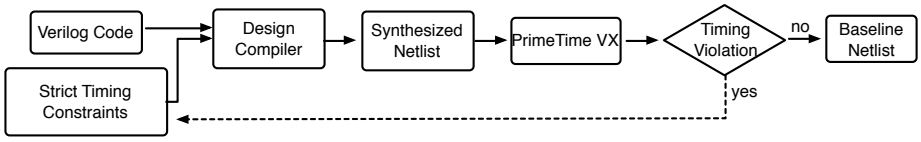
(b) Algorithm up-sizes the least number of gates in a circuit to reduce cost

**Figure 4.1:** (a) Part of the Safety Inference Analysis that finds precise wires. (b) Gate sizing algorithm for ASG approximate synthesis flow.

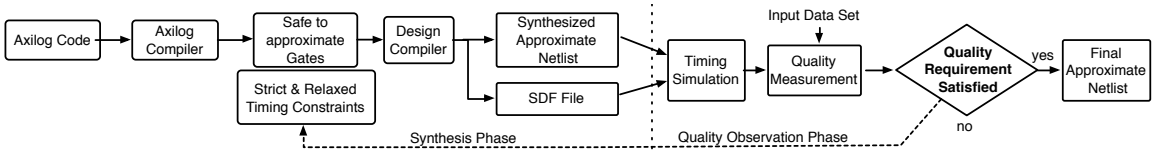
This procedure is a *backward-flow* analysis that has three phases: (1) The **first phase** identifies the *sink* wires, which are either unannotated outputs or wires *explicitly* annotated with restrict. The procedure then identifies the gates that are driving these sink wires and adds their input wires to the precise set. The algorithm repeats this step for the newly added wires until it reaches an input or an explicitly relaxed wire. However, this phase is only limited to the scope of the module-under-analysis; (2) **The second phase** identifies the relaxed outputs of the instantiated submodules. Due to the semantic differences between relax and relax_local, the output of a submodule will be considered relaxed if the following two conditions are satisfied. (a) The output drives another explicitly relaxed wire, which is not inferred due to a relax_local annotation; and (b) the output is not driving a wire already identified as precise. The algorithm automatically annotates these qualifying outputs as relaxed. The analysis repeats these two phases for all the instantiated submodules. For correct functionality of this analysis, all the module instantiations are distinct entities in the set $\mathbb{M}$ and are ordered hierarchically; (3) **In the final phase**, the algorithm marks any wire that affects a globally restricted wire as precise. Finally, the Safety Inference Analysis identifies the safe-to-approximate subset of the gates and wires with regards to the designer annotations. An approximation-aware synthesis tool can then generate an optimized netlist.

### 4.2.3 Approximate Synthesis

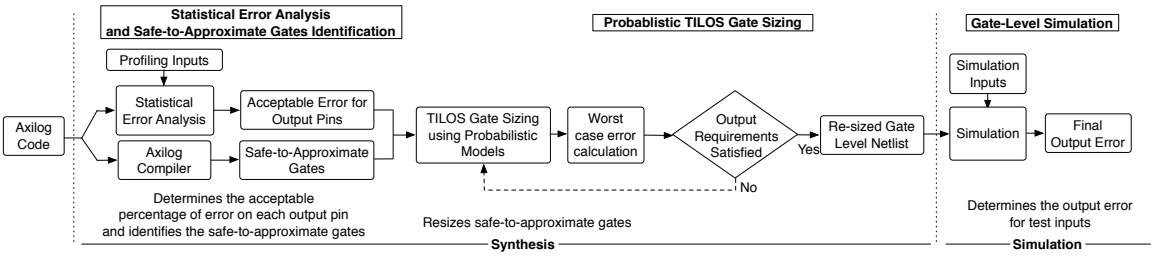
In our framework, approximate synthesis involves two stages. (1) In the **first stage**, annotated Verilog source code is converted to a precise gate-level netlist while preserving the approximate annotations. The Safety Inference Analysis then identifies the safe-to-approximate subset of the design based on designer annotations. (2) In the **second stage**, the synthesis tool applies approximate synthesis and optimization techniques *only* to the safe-to-approximate subset of the circuit elements. The tool has the liberty to apply any approximate optimization technique including gate substitution, gate elimination, logic restructuring, voltage over-scaling, and timing speculation as it deems prudent. The objec-




**(a)** Synthesis flow for baseline error-free gate netlist




**(b)** Synthesis flow for approximation by relaxing the timing (AST) for safe-to-approximate gates




**(c)** Synthesis flow for approximation using gate resizing (ASG) that uses probabilistic models as a proxy for future nodes

**Figure 4.2:** Synthesis flow for (a) baseline, (b) approximation using AST (c) approximation using ASG.

tive is to minimize a combination of error, delay, energy, and area considering final quality requirements. As Figure 4.2 shows, we developed two approximate synthesis flows to evaluate Axilog. In the next subsections we describe these flows in detail.

#### *4.2.3.1 AST: Approximate Synthesis through Relaxing Timing Constraints*

This synthesis flow is applicable to current technology nodes and leverages commercial synthesis tools. As shown in Figure 4.2a, we first use Synopsys Design Compiler to synthesize the design with no approximation. We perform a multi-objective optimization targeting the highest frequency while minimizing power and area. We will refer to the resulting netlist as the baseline netlist and its frequency as the baseline frequency. We account for variability using Synopsys PrimeTimeVX which, given timing constraints, provides the probability of timing violations due to variations. In case of violation, the synthesis process is repeated by adjusting timing constraints until PrimeTimeVX confirms

no violations. Second, as shown in Figure 4.2b, we only relax the timing constraints for the safe-to-approximate paths. We then extract the post-synthesis gate delay information in Standard Delay Format and perform gate-level timing simulations with a set of input datasets. We use the baseline frequency for the timing simulations even though some of the safe-to-approximate paths are synthesized with more timing slack. Timing simulations yield output values that may incur quality loss at the baseline frequency. We then measure the quality loss and if the quality loss is more than designer's requirements, we tighten the timing constraints on the safe-to-approximate paths. We repeat this step until the quality requirements are satisfied. This methodology has a potential to reduce energy and area by utilizing slower and smaller gates for the paths which use relaxed timing constraints.

#### *4.2.3.2 ASG: Approximate Synthesis through Gate Resizing*

The ASG synthesis flow studies the potential of approximate synthesis for future technology nodes. As the characteristics of transistors and gates for future technologies are unknown, we assume that the probability of error for a gate is an inverse function of its size. As a result, gate size, referred to as the PCMOS [131] area, should be treated as a proxy for the cost we would pay in a future technology node to get more robust gates. That cost could be, thicker gate oxides, higher threshold voltage and higher $V_{dd}$ to make the transistors more robust. The ASG synthesis flow applies approximation by selectively downsizing the gates as shown in Figure 4.2c. In this framework, smaller gates dissipate less energy and have smaller PCMOS area, however, may generate incorrect output with some probability. We now describe in detail the probabilistic error model for the gates.

**Probabilistic error models for gates.** Due to the unavailability of future nodes, we augment a currently available library–NanGate FreePDK 45 nm–with a probabilistic error model for all the gates the library. The error model provides the probability of a bit flip in the gate output. We use transistor-level SPICE simulations to find the probability of an

error at the gate output using the Cadence Virtuoso toolset. We take inspiration from the PCMOS models described in [131]. We simulated each gate at different sizes and the gate inputs were injected with Gaussian noise through a minimum-sized buffer. Gate error is also dependent on threshold voltage; however, we focused on gate sizing and its effects on gate error for a fixed threshold voltage. For each input combination, the noise was injected on gate inputs in the form of a Piece Wise Linear voltage source and the output was sampled for 10,000 inputs. Finally, the probability of correct output was computed as follows.

$$P_{\text{correct output}} = 1 - \frac{\text{Number of Incorrect Samples}}{\text{Total Number of Samples}} \tag{4.1}$$

We repeat this measurement for all the input combinations of the gate and assign the gate with the worst observed error. Next, we use this error model to optimize the power and area of the circuit by up-sizing the least number of gates in a circuit while satisfying the error requirements specified by the designer.

**Algorithm 2: Gate sizing optimization.** The ASG optimization algorithm shown in Figure 4.1(b) trades off accuracy for reduction in PCMOS area and energy. We extended the TILOS algorithm [132] to incorporate probabilistic models and changed the objective from minimizing delay to minimizing error and cost. The ASG optimization algorithm comprises of four phases.

**In the first phase** we extract the adjacency list (a space efficient way of representing a circuit) of the safe-to-approximate sub-circuit and determine its inputs and outputs.

**In the second phase** a Monte-Carlo simulation is used to determine the error-free probability of obtaining a 1 or a 0 at each node of the sub-circuit. For the Monte-Carlo simulation, random input vectors are applied to the inputs of the sub-circuit and a topological traversal propagates the values through the circuit for each input vector. This process gives us the probability of getting a 1 or 0 at the output of each gate. We then initialize all gates in the safe-to-approximate sub-circuit to their minimum size, i.e., having maximum error. We

calculate the initial error map $\gamma$ at the output of each gate by propagating the error through the circuit using the Boolean Error Propagation (BEP) algorithm [133]. The boolean error propagation algorithm then estimates the worst-case error probability for the outputs of the design based on each gates error probability model. If the calculated output error was not within the error requirements we entered phase 3.

In the **third phase**, for each safe-to-approximate output, we identify the gates that are driving that output, called the fan-in-cone, and add it to the fan-in hashmap $\beta$. In the **fourth phase**, for each gate in the fan-in-cone of safe-to-approximate output, we calculate the sensitivity of the output error to that gate. The sensitivity is measured by temporarily increasing the size of the gate to the next possible size and calculating the ratio of decrease in error to increase in gate size. Finally, after calculating the sensitivity for each fan-in gate we permanently up-size only the gate that shows the *largest impact* towards the output error. We perform the BEP using the changed gate size and update the error map $\gamma$. We repeat the fourth phase for each safe-to-approximate output, until user specified error bounds are satisfied for each safe-to-approximate output. The most compute intensive part of the algorithm is the Phase 3's boolean_error_propagation function with a complexity of $O(n^3)$. We optimized this function and reduced its complexity to $O(n^2)$ by decreasing the its iteration count by grouping gates together. These groups are resized together.

In the next section, we evaluate Axilog and the approximate synthesis processes with a set of benchmark designs.

### 4.2.4 Evaluation

**Benchmarks and Code Annotation.** Table 4.2 lists the Verilog benchmarks. We use Axilog annotations to judiciously relax some of the circuit elements. The benchmarks span a wide range of domains including arithmetic units, signal processing, robotics, machine learning, and image processing. Table 4.2 also includes the input datasets, application-specific quality metrics, the number of lines, and the number of Axilog annotations for

**Table 4.2:** Benchmarks, input datasets, and error metrics.

| Benchmark Name | Domain | Input Data Set | Quality Metric | # of Lines | # of Annotations | |
|---|---|---|---|---|---|---|
| | | | | | Design | Reuse |
| Brent-Kung (32-bit adder) | Arithmetic Computation | 1,000,000 32-bit integers | Avg Relative Error | 352 | 1 | 1 |
| FIR (8-bit FIR filter) | Signal Processing | 1,000,000 8-bit integers | Avg Relative Error | 113 | 6 | 5 |
| ForwardK (forward kinematics for 2-joint arm) | Robotics | 1,000,000 32-bit fixed-point values | Avg Relative Error | 18,282 | 5 | 4 |
| InverseK (inverse kinematics for 2-joint arm) | Robotics | 1,000,000 32-bit fixed-point values | Avg Relative Error | 22,407 | 8 | 4 |
| K-means (K-means clustering) | Machine Learning | 1024x1024-pixel color image | Image Diff | 10,985 | 7 | 3 |
| Kogge-Stone (32-bit adder) | Arithmetic Computation | 1,000,000 32-bit integers | Avg Relative Error | 353 | 1 | 1 |
| Wallace Tree (32-bit Multiplier) | Arithmetic Computation | 1,000,000 32-bit integers | Avg Relative Error | 13,928 | 5 | 3 |
| Neural Network (feedforward neural network) | Machine Learning | 1024x1024-pixel color image | Image Diff | 21,053 | 4 | 3 |
| Sobel (sobel edge dectector) | Image Processing | 1024x1024-pixel color image | Image Diff | 143 | 6 | 3 |

design and reuse.

**Axilog annotations.** We annotated the benchmarks with the Axilog extensions. The designs were either downloaded from open-source IP providers or developed without any initial annotations. After development, we analyzed the source Verilog codes to identify safe-to-approximate parts. The last two columns of Table 4.2 show the number of design and reuse annotations for each benchmark. The number of annotations range from 2 for Brent-Kung with 352 lines to 12 for InverseK with 22,407 lines. The Axilog framework enabled us to only use a handful of annotations to effectively approximate designs that are implemented with thousands of lines of Verilog.

The safe-to-approximate parts are more common in datapaths of the benchmarks rather than their control logic. For example, K-means involves a large number of multiplications and additions. We used the relax annotations to declare these arithmetic operations approximiable; however, we used restrict to ensure the precision of all the control signals. For smaller benchmarks, such as Brent-Kung, Kogge-Stone and Wallace Tree, only a subset of the least significant output bits were annotated to limit the quality loss. We also annotated the benchmarks with reuse annotations. The number of reuse annotations are listed in the last column of Table 4.2. Overall, one graduate student was able to annotate all the

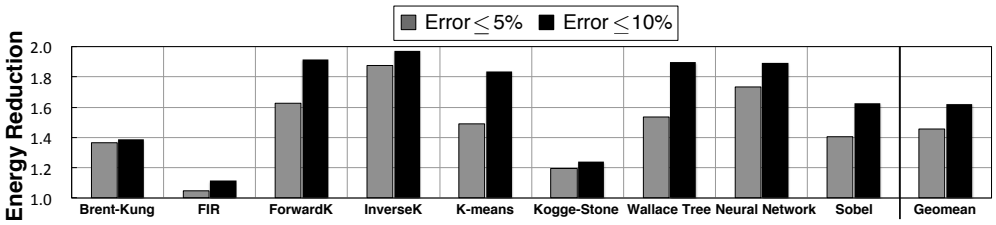
benchmarks within two days without being involved in their design. The intuitive nature of Axilog extensions makes annotating straightforward.

**Application-specific quality metrics.** Table 4.2 shows the application-specific error metrics to evaluate the quality loss due to approximation. Using application-specific quality metrics is commensurate with prior work on approximate computing and language design [134, 135]. In all cases, we compare the output of the original baseline application to the output of the approximated design.

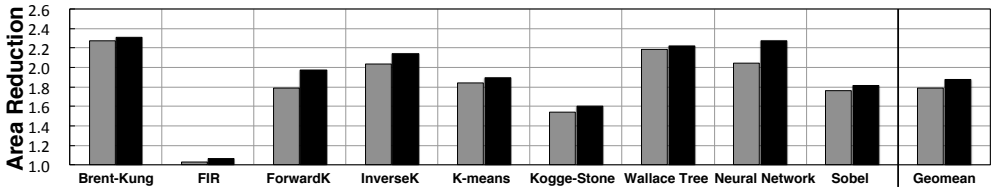
**Experimental results.** Both the synthesis techniques use Synopsys Design Compiler (G-2012.06-SP5) and Synopsys PrimeTime (F-2011.06-SP3-2) for synthesis flows and energy analysis, respectively.

**AST Evaluation.** We used Cadence NC-Verilog (11.10-s062) for timing simulation with SDF back annotations extracted from various operating corners. We use the TSMC 45-nm multi-$V_t$ standard cells libraries and the primary results are reported for the slowest PVT corner (SS, 0.81V, 0°C). The AST approach generates approximate netlists for the current technology node and provides, on average, 1.45× energy and 1.8× area reduction for the 5% limit. With the 10% limit, the average energy and area gains grow to 1.54× and 1.82× as shown in Figure 4.3a and 4.3b.

Benchmarks such as InverseK, Wallace Tree, Neural Network, and Sobel–that have a larger datapath–provide a larger scope for approximation and are usually the ones that see larger benefits. The structure of the circuit also affects the potential benefits. For instance, Brent-Kung and Kogge-Stone adders benefit differently from approximation due to the structural differences in their logic trees. The FIR benchmark shows the smallest energy savings since it is a relatively small design that does not provide many opportunities for approximation. Nevertheless, FIR still achieves 11% energy savings and 7% area reduction with 10% quality loss, suggesting that even designs with limited opportunities for approximation can




**(a)** Energy Reduction = (Precise circuit energy)/(Approximate circuit energy)

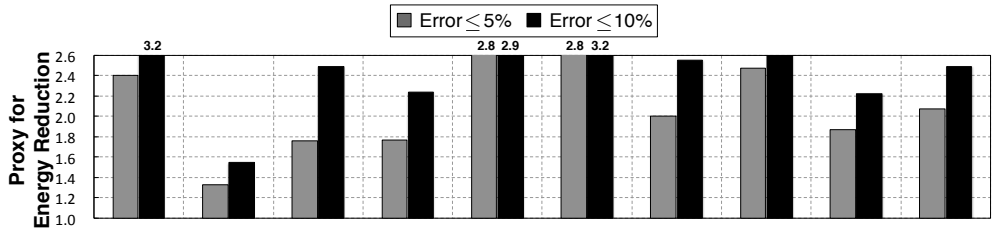



**(b)** Area Reduction = (Precise circuit area)/(Approximate circuit area)

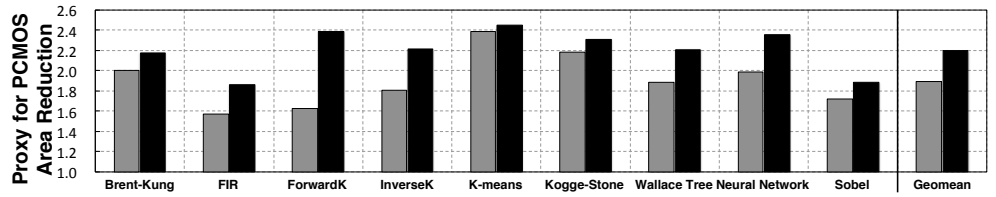
| PVT Corners | Brent-Kung | FIR | ForwardK | InverseK | K-means | Kogge-Stone | Wallace Tree | Neural Network | Sobel | Geomean |
|---|---|---|---|---|---|---|---|---|---|---|
| (SS, 0.81V, 0°C) | 34% | 11% | 78% | 87% | 69% | 24% | 65% | 83% | 57% | 54% |
| (SS, 0.81V, 125°C) | 32% | 7% | 72% | 79% | 65% | 21% | 63% | 72% | 41% | 48% |

**(c)** Energy reduction when the quality degradation limit is set to 10% for two different PVT corners. Here, we consider temperature variations.

**AST Synthesis Flow: reductions in (a) energy and (b) area when the quality degradation limit is set to 5% and 10%. (c) Energy reduction for two different PVT corners.**




**(d)** Proxy for Energy Reduction = (Precise circuit energy)/(Approximate circuit energy)




**(e)** Proxy for PCMOS area Reduction = (Precise circuit PCMOS area)/(Approximate circuit PCMOS area)

**ASG Synthesis Flow: reductions in (d) energy and (e) area when the quality degradation limit is set to 5% and 10% for the ASG synthesis flow.**

**Figure 4.3:** (a, b, c) Energy and Area reduction for AST flow. (d,e) Energy and PCMOS area reduction for ASG flow.

benefit significantly from Axilog.

We also evaluated the effectiveness of our AST technique in the presence of temperature variations for a full industrial range of 0°C to 125°C. We measured the impact of temperature fluctuations on the energy benefits for the same relaxed designs. Table 4.3c compares the energy benefits at the lower and higher temperatures (the quality loss limit is set to 10%). In this range of temperature variations, the average energy benefits ranges from 1.54× (at 0°C) to 1.48× (at 125°C). These results confirm the robustness of our framework; it yields significant benefits even when temperature varies.

**ASG Evaluation.** We used the NanGate FreePDK 45 nm multi-speed standard cells library. The AST and ASG techniques use different libraries because FreePDK 45 nm library allowed SPICE simulations required for the ASG flow. As mentioned before, the ASG flow aims to study the trends in future technology nodes when gates might show probabilistic behavior. We develop PCMOS models with the available libraries at 45 nm. The area numbers reported here are the ones set by the PCMOS model to satisfy the fixed gate robustness. These numbers do not necessarily correspond to actual area numbers in any future technology. The PCMOS area shows the relative cost savings across benchmarks and delineate the anticipated trends. As shown in Figures 4.3d and 4.3e, the ASG flow, provides, on average, 2× energy and 1.9× PCMOS area reduction for the 5% error limit. With the 10% limit, the average energy and area gains grow to 2.5× and 2.2×.

**Summary.** Both the synthesis frameworks show the effectiveness of Axilog and achieve significant savings while preserving the application functionality. This tradeoff is attainable because of the high-level language annotations and design abstractions allow the designer to target approximation where it is most effective without compromising the critical parts of the computation.

## 4.3 Related Work

A growing body of research shows the applicability and significant benefits of approximation [116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130]. However, prior research has not explored extending hardware description languages for systematic and reusable approximate hardware design. Below, we discuss the most related works.

**Approximate programming languages.** EnerJ [134] provides a set of type qualifier to manually annotate all the approximate variables in the program. If we had extended EnerJ's model to Verilog, the designer would have had to manually annotate all approximate wires/regs. Rely [135] asks for manually marking both approximate variables and operations, which requires more annotations. The work in [136] proposes language extension to the OpenMP software programming language that allows programmers to manually specify approximable regions of code. With our abstractions, the designer marks a few wires/regs and then the analysis automatically infers which other connections and gates are safe to approximate.

**Approximate circuit design and synthesis.** Prior work proposes imprecise implementations of custom instructions [137] and specific hardware blocks [118, 122, 119, 121, 123, 124]. The work in [125, 126, 127, 120, 128, 129, 130] propose algorithms for approximate synthesis that leverages gate pruning, timing speculation, or voltage overscaling. While all these synthesis techniques provide significant improvements, they do not focus on providing hardware description language semantics for methodical approximate hardware design and reuse. In fact, our framework can benefit and leverage all these synthesis techniques.

### 4.3.1 Conclusion

Axilog's automated analysis enables approximate hardware design and reuse without exposing the intricacies of synthesis and optimization. Furthermore, all the abstractions pre-

sented in this paper are concrete extensions to the mainstream Verilog HDL providing designers with backward compatibility. We evaluated Axilog, its automated Safety Inference Analysis, and presented two approximate synthesis techniques. Both flows demonstrate significant cost savings with merely 2 to 12 annotations per benchmark. These results confirm that Axilog is a methodical step toward practical approximate hardware design and reuse.

## 4.4 Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration

With the effective end of Dennard scaling [3], energy efficiency fundamentally limits microprocessor performance [1, 2]. As a result, there is a growing interest in specialization and acceleration, which trade generality for significant gains in performance and efficiency. Recent work shows three orders of magnitude improvement in efficiency and speed with Application Specific ICs [138]. However, designing ASICs for the massive and rapidly-evolving body of general-purpose applications is currently impractical. Programmable accelerators [10, 8, 7] establish a middle ground that exploit certain characteristics of the application to achieve performance and efficiency gains at the cost of generality. Tolerance to approximation is one such application characteristic. As the growing body of research in approximation shows, many application domains including web search, machine learning, multimedia, cyber-physical systems, vision, and speech recognition can tolerate small errors in computation [139, 140, 141, 142, 143, 144, 145, 146]. *Approximate accelerators* exploit this application characteristic by trading off computational accuracy for higher performance and better efficiency [147, 148, 149, 150, 151, 152, 153, 72]. Each invocation of the approximate accelerator improves performance and efficiency but may also lower the quality of the final output. The common practice in approximate acceleration is to *always* invoke the accelerator in lieu of a frequently-executed safe-to-approximate region of code,

e.g., a function in a loop. Always invoking the accelerator provides maximum gains from approximation but can potentially lead to an unacceptable *fixed degree* of quality loss. This approach does not provide the flexibility to explore the tradeoffs between quality and gains in performance and efficiency.

This paper tackles this shortcoming and defines a cohesively co-designed hardware-software solution, MITHRA, with components in both compiler and microarchitecture. [1]. MITHRA seeks to identify whether each individual accelerator invocation will lead to an undesirable quality loss. If MITHRA speculates that a large quality degradation is likely, it directs the processor to run the original precise code. This paper makes the following contributions by exploring the unique properties, challenges, and tradeoffs in designing MITHRA and defines solutions that address them.

**(1)** We find that a relatively low fraction of the accelerator invocations lead to large errors and need to be excluded from approximate acceleration. To accomplish the challenging task of filtering out this small subset and still delivering significant gains, we introduce MITHRA– a hardware-software co-design–for controlling the quality tradeoffs. The guiding principle behind MITHRA is that quality control is a binary classification task that either determines to invoke the approximate accelerator or the original precise code for each invocation.

**(2)** We devise MITHRA, the framework comprising of both hardware and software components. Hardware is devised to perform the binary classification task at runtime. The hardware also provides a knob to the software to control the quality tradeoffs. The software solves a statistical optimization problem to tune the knob. The solution provides statistical guarantees with a high confidence that desired quality loss levels will be met on unseen datasets. This tuned knob is used to pre-train the hardware classifiers.

**(3)** We evaluate MITHRA using an existing approximate accelerator [147] on a set of benchmarks with diverse error behaviors. We compare our designs with an ideal but infeasible

[1] Mithra is an angelic divinity which is a judicial figure and all-seeing protector of truth, and the guardian of cattle, the harvest, and the waters.

mechanism, referred to as *the oracle*. The oracle maximizes the performance and energy benefits by filtering out the fewest possible accelerator invocations for any level of final quality loss. We devise two realistic instances of hardware classifiers–table-based and neural network based– that aim to mimic the behavior of the oracle. The results show that both of our designs closely follow the oracle in delivering performance and efficiency gains.

For 5% quality loss, with 95% confidence and 90% success rate, the table-based design with eight tables each of size 0.5 KB achieves 2.5× average speedup and 2.6× average energy reduction. The neural design shows similar speedup, however provides 13% better energy reduction. Compared to the table-based design, the ideal oracle with prior knowledge about all invocations achieves only 26% higher speedup and 36% better energy efficiency. Compared to the neural design, the oracle achieves 26% and 19% higher performance and efficiency benefits, respectively. These results suggest that MITHRA makes an effective stride in providing hardware-software solutions for controlling quality tradeoffs. Such solutions are imperative in making approximate acceleration widely applicable.

### 4.4.1 Challenges and Overview

Approximate accelerators trade small losses in output quality for significant performance and efficiency gains [147, 149, 148, 150, 151, 152, 153, 72]. When a processor core is augmented with an approximate accelerator, the core delegates the computation of frequently executed safe-to-approximate functions to the accelerator. A safe-to-approximate function is a region of code that can tolerate imprecise execution. Instead of executing the function, the core sends the function's inputs to the accelerator and retrieves its outputs. The outputs from the accelerator are an approximation of the outputs that the core would have calculated by executing the original function. The configuration of the accelerator is generated by the compiler.

The accelerator is always invoked in lieu of the original function. Always invoking the accelerator leads to a *fixed* tradeoff between quality and gains in performance and effi-

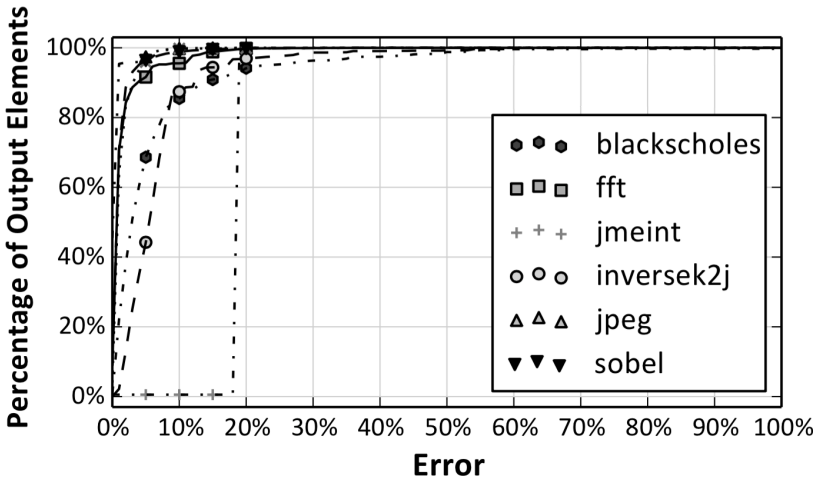



**Figure 4.4:** Cumulative distribution function plot of the applications output error. A point $(x, y)$ implies that $y$ fraction of the output elements see error less than or equal to $x$ [147].

ciency. The lack of flexibility in controlling this tradeoff limits the applicability of approximate acceleration. Therefore, we devise MITHRA, a hardware-software solution that can effectively control the quality tradeoffs.

#### *4.4.1.1 Challenges and Insights*

MITHRA's objective is to provide flexibility in controlling final quality loss and to maximize the performance and energy benefits at any level of quality. MITHRA aims to only filter out those approximate accelerator invocations that cause relatively large quality degradation in the final output. To devise such a solution, we analyze the properties and challenges of a system augmented with an approximate accelerator. Below we discuss the insights that guide the design of MITHRA.

**1) What accelerator characteristics can guide the design of MITHRA?** We investigate the error distribution in the output of different applications when they undergo approximate acceleration [147]. Figure 4.4 shows the cumulative distribution function (CDF) plot of final error for each element of the applications output. The application output is a collection of elements, e.g., image consists of pixels, vector consists of scalars, etc. As illustrated in the Figure 4.4, only a small fraction (0%-20%) of these elements see large errors. The

main insight is that there is an opportunity to attain significant gains with approximate acceleration while reducing the quality loss. To exploit this opportunity, MITHRA filters out these small fraction of accelerator inputs that lead to relatively large errors.

**2) What information is needed and is available to segregate accelerator inputs that cause large error?** For each invocation, the core sends a set of inputs (the input vector) to the accelerator and retrieves a set of outputs (the output vector). The input vector here refers to the accelerator input and not the application input. The accelerator output is the function of its input vector and its configuration. The difference between the imprecise accelerator output and the precise output is the accelerator error. Therefore, accelerator error becomes a function of the input vector and the accelerator configuration. For a given application, the accelerator configuration is fixed; making the accelerator error only a function of the input vector. Accelerator inputs provide enough information to determine whether or not a specific accelerator invocation will lead to a relatively large error. This insight simplifies the design of MITHRA, enabling it to use only information that is local to each invocation.

**3) How to map the final output quality loss as a local accelerator error?** MITHRA uses the accelerator inputs to classify accelerator invocations that cause large error. Therefore, MITHRA makes local decisions based on the accelerator input without a global view of execution and without any knowledge about the manifestation of the accelerator error on the final output. The main challenge is to devise a technique that can find an upper bound on the accelerator error to guide MITHRA's local decisions while considering the final output. To address these challenges, we develop a statistical optimization solver that finds an optimal threshold for the accelerator error. This threshold maximizes accelerator invocations and gains from approximation while providing statistical guarantees that the final quality loss level will be met with high confidence. This mechanism tries to keep the accelerator error below a certain threshold for each invocation. MITHRA considers the local error large if it speculates that any element in the accelerator output vector might have an error larger than the threshold. The threshold forms the knob for controlling the quality tradeoffs.

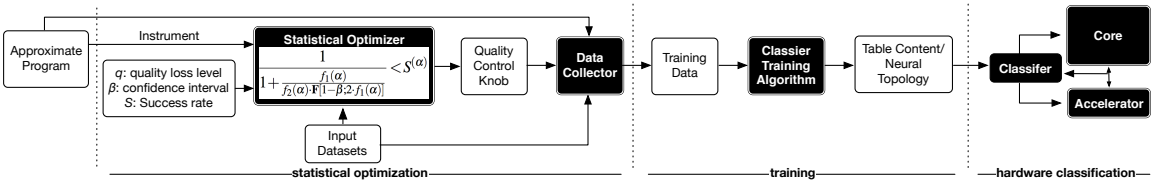



**Figure 4.5:** Overview of MITHRA comprising a statistical optimizer, a trainer, and a hardware classifier. The statistical optimizer uses instrumented approximate program and input datasets to tune the quality control knob such that it satisfies a desired quality loss ($q$) with high confidence ($\beta$) and success rate ($S$). The knob is used to generate the training data for the classifiers. The classifiers operate at runtime to control the quality tradeoffs.

**4) What guarantees are provided for the final output quality loss?** The final output quality is mapped onto the local accelerator site as the threshold for the accelerator error. The final quality loss is provided by the programmer requiring either formal or statistical guarantees. In general, providing formal guarantees that the quality requirements will be met on all possible application input datasets is intractable due to the complex behavior of programs and large space of possible inputs. Statistical approach is the most viable alternative to validate quality control techniques. Therefore, we incorporate the Clopper-Pearson exact method to provide statistical guarantees that the desired level of quality loss will be met with high confidence on unseen datasets.

**5) What needs to be done for MITHRA in hardware?** MITHRA's objective is to identify accelerator invocations that lead to a relatively large quality loss or an error above the threshold. Measuring the error requires running both the original precise code and invoking the accelerator, which will nullify the benefits of acceleration. MITHRA makes this decision without measuring the actual error of the accelerator. This decision making can be accomplished by using a classification algorithm that maps the accelerator inputs to a binary decision. The main challenge is to devise classifiers that can be efficiently implemented in hardware and make quality tradeoff decisions at runtime.

### *4.4.1.2 Overview*

MITHRA's framework, shown in Figure 4.5, comprises (1) a compiler constituting statis-

tical optimizer and pre-trainer, and (2) a hardware classifier which is a microarchitectural mechanism that resides between the core and the accelerator. The hardware classifier uses the accelerator inputs to make a binary decision of either running the original function on the core or invoking the accelerator. In this paper, we propose and explore two classification algorithms that can be implemented as the microarchitectural instances of MITHRA. The first algorithm is a novel table-based mechanism that efficiently hashes the accelerator input vector to retrieve the decision from a small ensemble of tables. The second technique is a multi-layer perceptron based neural mechanism. Section 4.4.3 describes these two hardware classifiers.

The hardware classifiers need to be trained in order to make decisions at runtime. The compilation component of MITHRA trains these classifiers to detect accelerator inputs that might produce accelerator error greater than a threshold. This threshold is the quality control knob that is tightened or loosened in accordance to the desired level of final output quality. Optimal threshold is obtained by solving a statistical optimization problem that maximizes accelerator invocation rate and benefits from approximate acceleration for a set of representative application input datasets while keeping the quality loss below a desired level. The optimization provides statistical guarantees that final quality loss will be met with high confidence and success rate. The thresholding mechanism and pre-training is described in Section 4.4.2.

### 4.4.2 Statistical Optimization for Controlling Quality Tradeoffs

Approximate accelerators require the programmer to provide an application-specific quality metric and a set of representative input datasets [147, 149, 148, 150, 151, 152, 153, 72]. MITHRA uses the same information to automatically train the classifiers. This training process constitutes two phases. The first phase, referred to as the *thresholding phase* utilizes profiling information to find a threshold for the accelerator error. This phase converts the global quality loss into a local accelerator error threshold by solving a statistical

optimization problem.

The second phase or the *training phase*, generates training information for the hardware classifiers based on the threshold found in the first phase (Section 4.4.2.2). This training information is incorporated in the accelerator configuration and is loaded in the classifiers when the program is loaded to the memory for execution. This strategy is commensurate with previous works on acceleration (precise or approximate) that generate the accelerator configuration at compilation time and encode it in the binary [147, 148, 149, 8, 7]. The configurations of both the accelerator and MITHRA are part of the architectural state. Therefore, the operating system must save and restore the configuration data for both the accelerator and MITHRA on a context switch. To reduce context switch overheads, the OS can use the same lazy context switch techniques that are typically used with floating point units [154].

#### *4.4.2.1 Finding the Threshold*

The objective of this phase is to tune the quality control knob, i.e., find a threshold for the accelerator error. The threshold enables MITHRA to maximize the accelerator invocations for any level of quality loss. The optimized threshold is an upper bound on the error that can be tolerated by the target function from the accelerator to maintain the desired final output quality. To allow an accelerator invocation, the error of each element of the output vector should be below the threshold (*th*), as shown in Equation 4.2.

$$\forall o_i \in OutputVector \quad |o_i(precise) - o_i(approximate)| \leq th \tag{4.2}$$

Algorithm 1 shows the iterative procedure to find this optimized threshold. In this process, for each intermediate threshold (*th*), the program (P) can be instrumented to find the final quality loss ($q_i$) for a set of representative application input datasets ($i$). Each final quality loss level ($q_i$) is compared with the desired final quality loss($q$). Due to the complex behavior of programs and the large space of possible datasets, some application inputs

might fall within the desired final quality loss, while the others might not. Hence, providing formal guarantees that quality requirements will be met on all possible application inputs is intractable. Statistical approaches are the most viable solutions to validate quality control techniques. Therefore, MITHRA provides statistical guarantees that quality requirements will be met on unseen datasets with high confidence. To provide such guarantees, the algorithm counts the application input datasets that have final output quality ($q_i$) below or equal to the desired quality loss ($q$) as shown in Equation 4.3.

$$\forall i \in inputSet \quad if(q_i(P_i(th)) \leq q) \quad n_{success} = n_{success} + 1 \tag{4.3}$$

The number of application outputs that have desired quality loss ($n_{success}$) varies with the threshold ($th$). For instance, as the threshold is made tighter the number of $n_{success}$ will increase as the output quality loss of each application input ($q_i$) will decrease. We utilize the $n_{success}$ to compute the binomial proportion confidence interval and success rate using Clopper-Pearson exact method [155] as described below.

**Clopper-Pearson exact method.** As Equation 4.4 shows, the Clopper-Pearson exact method computes the one-sided confidence interval of success rate $S^{(q)}$, when the number of application inputs or sample trials, $n_{trials}$, and the number of successes among the trials, $n_{success}$, are measured for a sample of the population. In Equation 4.4, F is the F-critical value that is calculated based on the F-distribution [156].

$$\frac{1}{1 + \frac{n_{trials} - n_{success} + 1}{n_{success} \cdot \mathbf{F}[1-\beta; 2 \cdot n_{success}, 2 \cdot (n_{trials} - n_{success} + 1)]}} < S^{(q)} \tag{4.4}$$

To further understand Equation 4.4 and the $S^{(q)}$, we discuss a simple example in which, 90 ($n_{success}$) out of the total 100 ($n_{trials}$) representative application input datasets generate outputs that have a final quality loss $\leq 2.5\%$. In this example, the lower limit of the 95% confidence interval ($S^{97.5\%}$) is 80.7%. This implies that with 95% confidence we can project that at least 80.7% of unseen input sets will produce outputs that have quality

loss level within 2.5%. This projection is conservative because the Clopper-Pearson exact method calculates a conservative lower bound for the confidence interval. The degree of confidence ($\beta$) is the probability of the projection being true. The projection based on 95% confidence interval is true with probability of 0.95. The statistical optimization algorithm incorporates this Clopper-Pearson exact method. The optimization iteratively searches for an optimal threshold that maximizes accelerator invocations while providing high confidence that final quality loss will be met on unseen datasets.

```
Input  : P: Program
         ρ: Input data sets
         D: Quality loss with 100% accelerator invocation
         q: Desired quality loss level
         β: Confidence interval
         S: Desired success rate
Function SuccessRate (q, P, ρ, β, th)
    Initialize num ← 0
    for (∀ ρ_i in ρ) do
        P_i = Instrument (P)
        error = RunMeasureQuality (P_i, ρ_i, th)
        if (error ¡ q) then
            num = num + 1;
    end
    ClopperPearson (num, β, ρ)
    return num
end
Initialize th ← th_0
θ = SuccessRate (q, P, ρ, β, th)
terminate = false
while (terminate == false) do
    if (θ ¡ S) then
        th = th - Δ
    else if (θ ¿ S) then
        th_last = th
        th = th + Δ
    θ_last = θ
    θ = SuccessRate (q, P, ρ, β, th)
    if (θ ¡ S and θ_last ¿ S) then
        terminate = true
        return th_last
end
```

**Algorithm 2:** Finding the threshold.

The inputs to the algorithm are the program code ($P$), set of representative input datasets ($\rho$), quality degradation when accelerator is always invoked ($\mathcal{D}$), desired quality loss level ($q$), confidence interval ($\beta$) and desired success rate ($S$). The algorithm goes

through following steps:

**(1) Initialize.** Assign a random value to the threshold.

**(2) Instrument.** Instrument the program to execute both the original function and the accelerator for all invocations of the target function. For each invocation, use the original precise result if the accelerator error exceeds the threshold.

**(3) Measure the quality.** Run the instrumented program for each application input and measure the final output quality.

**(4) Measure the success rate.** Calculate the number of inputs that have the final output quality within the desired level ($q$). Use this number and confidence interval to calculate the success rate ($\theta$) with the Clopper-Pearson exact method.

**(5) Adjust the threshold.** If the success rate ($\theta$) is less than $S$, decrease the threshold by a small delta. If the success rate ($\theta$) is greater than $S$, increase the threshold by a small delta.

**(6) Reiterate or terminate.** Terminate if success rate ($\theta$) with the last threshold is greater than $S$ and with the current threshold is less than $S$. Otherwise, go to step (2).

As Section 4.4.6 elaborates, we use a different set of input datasets to validate the selection of the threshold. If the application offloads multiple functions to the accelerator, this algorithm can be extended to greedily find a tuple of thresholds. Due to the complexity of application behavior, this greedy approach will find suboptimal thresholds if the number of offloaded functions increases. After finding the threshold, the compiler profiles application input datasets to generate the training data for the classifiers.

#### *4.4.2.2 Training Data for Hardware Classifiers*

Once the threshold is determined, hardware classifiers can be pre-trained using representative application inputs. Generating training data requires running the application and randomly sampling the accelerator error. For the sampled invocations, if the accelerator error for all elements in the output vector are less than the threshold, the corresponding input is mapped to invoke the accelerator (binary decision '0'). Conversely, the input vector is

mapped to trigger the execution of the original function (binary decision '1') if the accelerator error is greater than the threshold. The training data is a collection of tuples. Each tuple contains an input vector and a binary value. The binary value is 0 if the accelerator error is larger than the threshold and 1 otherwise. For a given input vector, this binary value is the function of accelerator configuration and the threshold. Both the configuration and the threshold are constant for a fixed application and final quality loss. Therefore, a set of representative accelerator input vectors is sufficient to generate the training data. In many cases, a small number of application input datasets is sufficient to generate this training data for classifiers because the target function is hot and is executed frequently in a single application run. For instance, in an image processing application, the target function runs for every pixel in the input image. Even a single 512×512 input image provides 262,144 training data points. The generated training data is agnostic to the type of hardware classifier that needs to be trained. However, the training process depends on classifier. In this paper, we focus on a table-based and neural network based classifier. These classifiers and their training process is detailed in the next section.

### 4.4.3 Designing Hardware Classifiers for MITHRA

This section provides details about how hardware classifiers of MITHRA are designed to be deployed at runtime. MITHRA's microarchitectural component is a hardware classifier that maps an accelerator input vector with multiple elements to a single-bit binary decision. This binary decision determines whether MITHRA would invoke the accelerator or run the original precise function. This section defines and explores two hardware classifiers for MITHRA, one table-based and one based on neural networks. The table-based classifier mostly utilizes storage for decision making, whereas the neural classifier relies on arithmetic operations.

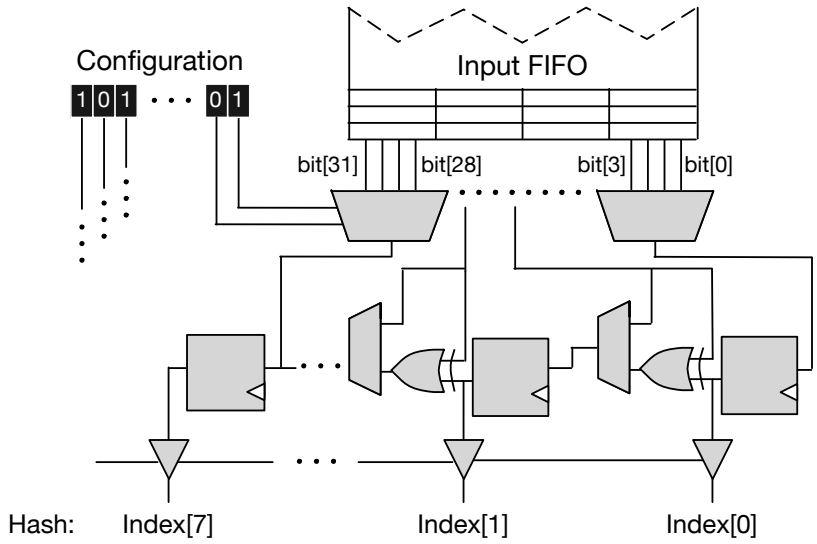



**Figure 4.6:** A reconfigurable hash function. Each hash function takes an input vector and generates the index. All the hashes are MISRs but the configuration register decides the input bits they use.

#### *4.4.3.1 Table-Based Classifier Design*

We devise a novel table-based classifier that stores its decisions (single-bit values) in a table, which are indexed by a hash over the elements of the accelerator input vector. We design an efficient circuit to hash the input elements and generate the index aiming to minimize aliasing. Below, we first discuss the hash function and then describe a multi-table design that leverages a small ensemble of tables to achieve better accuracy with limited storage.

**Generating Index from Multiple Inputs** For the table-based design, the hash function should (1) be able to combine all the elements in the input vector, (2) be able to reduce destructive aliasing as much as possible, (3) be efficiently implementable in hardware, (4) be able to accept a varying number of inputs, and, (5) be reconfigurable to work across different applications. To efficiently satisfy these requirements, we use a hardware structure called Multi-Input Signature Register (MISR) [157] to hash the input elements and generate the table index. A MISR takes in a bit-vector and uses a set of XOR gates to combine the incoming bit-vector with the content of a shift register. Figure 4.6 illustrates an instance of our MISR hash function. In a MISR, the result of the XORs is stored in a register

after a shift operation. As the next input comes in, the MISR repeats the previous step of combining the inputs together. After all the elements of the input vector for a single invocation are processed, the value that remains in the register is the index.

Number of elements in the accelerator input vector vary across applications. Therefore, the hash function should be able to accept a varying number of inputs. MISRs can naturally combine arbitrary number of input elements. As Figure 4.6 illustrates, we designed a reconfigurable MISR that supports different combinations of XOR, bit selection, and shift operations to allow the table-based classifier to adapt to the needs of the application. This configuration is decided at compile time for each application and is fixed during execution. The fixed hashing for each application makes the indexing completely deterministic.

One of the challenges with using MISRs to index the tables is that its content changes as the input elements arrive. These transient changes in the index of the tables can cause high energy consumption. Therefore, we connect the MISRs through a series of tri-state gates to the tables. The tri-state gates are inactive until the arrival of the last input element, to prevent the transient state of the MISRs from affecting the table. A counter counts the number of received input elements and activates the tri-state gates when the last element arrives. We send accelerator inputs to both the accelerator and the classifier simultaneously assuming that in most cases the classifier will decide to invoke the accelerator. This strategy is in line with the earlier insight that only a small fraction of the invocations require invoking the original precise code.

**Multi-Table Classifier** A single-table requires a large number of entries to maximize the benefits of approximate acceleration while reducing quality loss. The reason being that only a small fraction of the input combinations need to be filtered out from accelerator invocation. This characteristic makes it harder for only one small table to segregate inputs that cause relatively large errors because of destructive aliasing. When aliasing occurs and multiple inputs collide on the same entry, the bias is always toward invoking the accelerator.

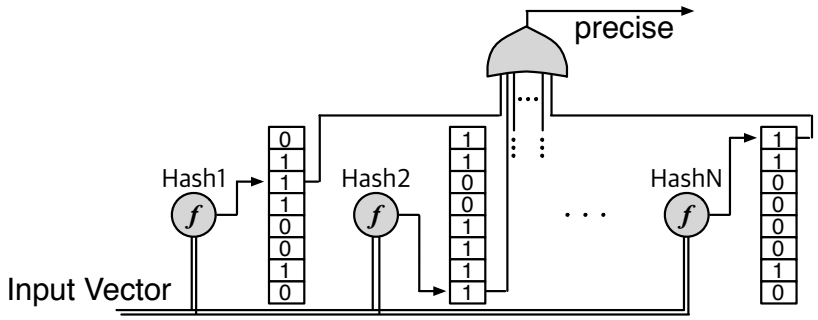



**Figure 4.7:** Multi-table based Classifier. All tables are equally sized but each table is indexed with a different MISR or hash configuration.

This bias may impair the ability of the smaller tables to distinguish the inputs, thereby causing relatively large quality losses.

To address this issue, we devise a multi-table classifier illustrated in Figure 4.7. The design consists of multiple equally-sized tables and each entry in the table is a single-bit value. The hash function for each table is a different MISR configuration. These configurations are selected from a pool of 16 fixed MISR configurations that exhibit least similarity, i.e., they map same input to different table indices. This configuration pool is independent of the application. The compiler assigns the first table with the MISR configuration that incurs least aliasing. The second table is assigned a different MISR that has least amount of aliasing and the combination of the two tables provides least false decisions. The compiler repeats this step for the third, fourth, etc., tables. We developed this greedy algorithm since the decision space is exponentially large. Using different hash functions for each table lowers the probability of destructive aliasing in the tables. As the input elements arrive, all the MISRs generate indices in parallel and the corresponding values are read from the tables. Since the bias in each single table is toward invoking the accelerator, MITHRA directs the core to run the original function even if a single table determines that the precise code should be executed. Therefore, the logic for combining the result of the tables is just an OR gate. The multi-table design is similar to Boosting in machine learning that combines an ensemble of weak learners to build a powerful classifier.

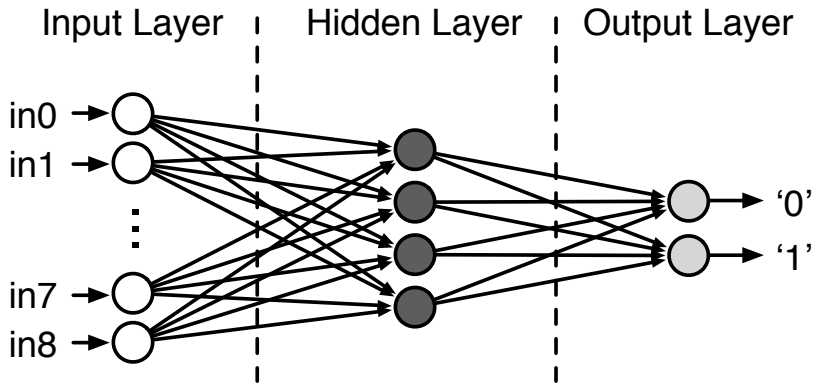



**Figure 4.8:** The neural classifier takes in accelerator inputs and generates two outputs. The output neuron with the larger value is the final outcome.

#### *4.4.3.2 Neural Classifier Design*

We also explore the use of neural networks to control the quality tradeoffs. While the table-based design utilizes storage for controlling the quality tradeoffs, the neural design leverages arithmetic operations for the same task. The neural classifier spends some of the gains achieved in performance and efficiency to obtain a higher quality in the results.

We use multi-layer perceptrons (MLPs) due to their broad applicability. An MLP consists of a fully-connected set of neurons organized into layers: the input layer, any number of hidden layers, and the output layer (Figure 4.8). A larger, more complex network offers greater accuracy, but is likely to be slower and dissipate more energy. To strike a balance between accuracy and efficiency, we limit neural design to three layer networks comprising one input layer, one hidden layer, and one output layer. Furthermore, we only consider neural networks with 2, 4, 8, 16, and 32 neurons in the hidden layer even though more neurons-per-layer are possible. The neural classifier takes in the same number of inputs as the accelerator and always contains two neurons in the output layer. One neuron represents the output ‘0’ and the other represent the output ‘1’. The output neuron with the larger value determines the final decision. We train [158] these five topologies and choose the one that provides the highest accuracy with the fewest neurons. During an invocation, as the core sends the input elements to neural network, which is executed on a specialized hardware

and decides whether or not to invoke the accelerator. The next subsection describes how these hardware classifiers are trained using the threshold specific training data.

### 4.4.4 Training the Classifiers

Table-based classifier is pre-trained offline. At runtime the table-based design is updated as we discuss later in this section. Updating the neural design online requires more computation power and may incur significant overheads. Therefore, for the neural design we follow the same workflow as the neural processing units (NPUs) [147, 72, 148, 149, 153] and train the neural network offline.

#### *4.4.4.1 Training the Table-Based Classifier*

The offline training of the table-based design initially sets all the table entries to '0' enabling a 100% accelerator invocation. For each training tuple, we calculate the hash for each accelerator input vector to identify its corresponding table entry. If a particular accelerator input vector leads to an error larger than the threshold, the corresponding table entry is set to '1'. In the case of aliasing, the table entry will be set to '1' even if only one of the aliased inputs results in an error larger than the threshold. This training strategy is conservative and avoids the bias towards invoking the accelerator since most of the accelerator inputs lead to small errors. The same procedure is extrapolated to train the ensemble of tables. After pre-training, we compress the content of these tables using the Base-Delta-Immediate compression algorithm [159] and encode the compressed values in the binary.

**Online training for the table-based design.** After deploying the pre-trained table-based design, we use the runtime information to further improve its accuracy. We sample the accelerator error by running both the original precise code and the accelerator at sporadic intervals. After sampling the error, the table entry is updated according to the same proce-

dure used in pre-training. In addition to generating the hash and updating the table entries, the online table update requires a few arithmetic operations to calculate the error and compare it with the threshold.

#### *4.4.4.2 Training the Neural Network Design*

Similar to the prior works [147, 72, 148, 149, 153], we use offline training the neural classifier. An alternative design could train the neural design concurrently with in-vivo operation. Online training could improve accuracy but would result in runtime overheads. To mitigate these overheads, an online training system could offload neural training to a remote server on the cloud.

### 4.4.5 Instruction Set Architecture Support

We add a special branch instruction to the ISA that invokes the original code if MITHRA decides to fall back to the precise code. Hence, the branch is taken if the hardware classifier speculates that the original precise function should be executed. This special branch instruction is inserted after the instructions that send the inputs to the accelerator. The overhead of this instruction is modeled in our evaluations.

### 4.4.6 Evaluation

#### *4.4.6.1 Experimental Setup*

**Cycle-accurate simulation.** We use the MARSSx86 x86-64 cycle-accurate simulator [160] to measure the performance of the accelerated system augmented with MITHRA. The processor is modeled after a single-core Intel Nehalem to evaluate the performance benefits over an aggressive out-of-order architecture[2]. We use NPU [147] as the approxi-

[2]**Processor:** Fetch/Issue Width: 4/6, INT-ALUs/FPUs: 3/2, Load/Store FUs: 2/2, ROB Size: 128, Issue Queue Size: 36, INT/FP Physical Registers: 256/256, Branch Predictor: Tournament 48KB, BTB Sets/Ways: 1024/4, RAS Entries: 64, Load/Store Queue Entries: 48/48, Dependence Predictor: 4096 Bloom Filter,

mate accelerator to evaluate MITHRA. The NPU consists of eight processing elements that expose three queues to the processor to communicate inputs, outputs, and configurations. The simulator is modified to include ISA-level support for the NPU[3]. This support consists of two enqueue and two dequeue instructions and a special branch instruction for executing the original function. The processor uses the same architectural interface and FIFOs to communicate the configuration of classifiers. Classifiers receive the inputs as the processor enqueues them in the accelerator FIFO. We use GCC v4.7.3 with -O3 to enable compiler optimization. The baseline in our experiments is the benchmark run solely on the processor with no approximate acceleration.

We augmented MARSSx86 with a cycle-accurate NPU simulator that also models the overheads of hardware classifiers. For the table-based mechanism, these overheads include, cycles to decompress the content, generate the indices, index into the table, and finally generate the decision. We use the NPU to execute the neural design which adds extra cycles and energy to the overall system.

**Energy modeling.** We use McPAT [161] for processor energy estimations. We model the NPU energy using results from McPAT, CACTI 6.5 [162], and [163]. The cycle-accurate NPU simulator provides detailed statistics that we use to estimate the energy of both the accelerator and the neural classifier. For estimating the energy of the table-based design, we implement the MISRs in Verilog and synthesize them using Synopsys Design Compiler (G-2012.06-SP5). We use Synopsys PrimeTime (F-2011.06-SP3-2) to measure the energy cost of the MISRs after synthesis. The synthesis library is the NanGate 45 nm Open Cell Library– an open source standard cell library. We also use the same synthesis procedure to measure the cost of arithmetic operations that are required to decompress the content in each table. We use CACTI 6.5 to measure the energy cost of accessing the tables. The processor, hard-

ITLB/DTLB Entries: 128/256 **L1:** 32KB Instruction, 32KB Data, Line Width: 64bytes, 8-Way, Latency: 3 cycles **L2:** 2MB, Line Width: 64bytes, 8-Way, Latency: 12 cycles **Memory Latency:** 50 ns

[3] **NPU**: Number of PEs: 8, Bus Schedule FIFO: 512x20-bit, Input FIFO: 128x32-bit, Output FIFO: 128x32-bit, Config FIFO: 8x32-bit **NPU PE**: Weight Cache: 512x33-bit, Input FIFO: 8x32-bit, Output Reg File: 8x32-bit, Sigmoid LUT: 2048x32-bit, Multiply-Add Unit: 32-bit Single-Precision FP

**Table 4.3:** Size of compressed table-based and neural classifiers.

| Benchmark | Size of Table-based Design after Compression (KB) | Neural-base Design | |
|---|---|---|---|
| | | Size (KB) | Neural Topology |
| **blackscholes** | 0.25 | 0.57 | 6->4->2 |
| **fft** | 0.25 | 0.10 | 1->4->2 |
| **inversek2j** | 0.29 | 0.10 | 2->4->2 |
| **jmeint** | 0.25 | 1.47 | 18->16->2 |
| **jpeg** | 3.70 | 0.79 | 64->2->2 |
| **sobel** | 3.20 | 0.22 | 9->4->2 |

**Table 4.4:** Benchmarks, their quality metric, input data sets, and the initial quality loss when the accelerator is invoked all the time.

| Benchmark | Description | Type | Application Error Metric | Input Data | Compilation Dataset | Validation Dataset | NPU Topology | Error with Full Approximation |
|---|---|---|---|---|---|---|---|---|
| **blackscholes** | Math model of a financial market | Financial Analysis | Avg. Relative Error | 4096 Data Point from PARSEC Suite | 250 Distinct | 250 Distinct | 6->8->8->1 | 6.03% |
| **fft** | Radix-2 Cooley-Tukey fast fourier | Signal Processing | Avg. Relative Error | 2048 Floating Point Numbers | 250 Distinct | 250 Distinct | 1->4->4->2 | 7.22% |
| **inversek2j** | Inverse kinematics for 2-joint arm | Robotics | Avg. Relative Error | 10000 (x, y) Coordinates | 250 Distinct | 250 Distinct | 2->8->2 | 7.50% |
| **jmeint** | Triangle intersection detection | 3D Gaming | Miss Rate | 10000 Pairs of 3D Triangle Coordinates | 250 Distinct | 250 Distinct | 18->32->8->2 | 17.69% |
| **jpeg** | JPEG encoding | Compression | Image Diff | 512x512-Pixel Color Image | 250 Distinct | 250 Distinct | 64->16->64 | 7.00% |
| **sobel** | Sobel edge detector | Image Processing | Image Diff | 512x512-Pixel Color Image | 250 Distinct | 250 Distinct | 9->8->1 | 9.96% |

ware classifier, and the accelerator operate at 2080 MHz at 0.9 V and are modeled at 45 nm technology node. These settings are in line with the energy results in [163] and [147].

**Classifier configurations.** For the main results in this section, we use a table-based design that consists of eight tables, each of size 0.5 KB. This design is the result of our Pareto analysis, presented in Figure 4.14. The topology of the neural classifier varies across benchmarks (Table 4.3).

**Benchmarks.** We use AxBench, a publicly available benchmark suite (`http://www.axbench.org`) that is used in [147, 148]. These benchmarks come with NPU topology and we use them without making any NPU-specific optimizations for utilizing MITHRA. These benchmarks represent a diverse set of application domains, including financial analysis, signal processing, robotics, 3D gaming, compression, and image processing. Table 4.4 summarizes each benchmark's application domain, input data, NPU topology, and final ap-

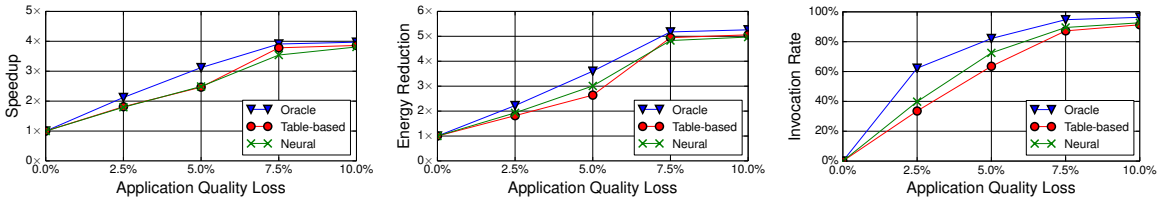
plication error levels when the accelerator is invoked for all inputs without MITHRA. We use each benchmark's application-specific error metric to evaluate MITHRA. The initial error with no quality control and full approximation ranges from 6.03% to 17.69%. This relatively low initial error makes the quality control more challenging and the diversity of the application error behavior provides an appropriate ground for understanding the tradeoffs in controlling quality with MITHRA.

**Input datasets.** We use 250 distinct datasets during compilation to find the threshold and train MITHRA. We use 250 different unseen datasets for validation and final evaluations that are reported in this section. Each dataset is a separate typical program input, such as a complete image (see Table 4.4).

#### *4.4.6.2 Experimental Results*

In this paper, we devise an optimized hardware for the table-based and neural based classifiers, and provide the necessary microarchitectural support. The table-based and neural based designs can also be implemented in software. To justify the hardware implementation of these classifiers we implement these algorithms in software and measure the corresponding application runtimes. The software implementation of the table-based and neural classifiers slow the average execution time by 2.9× and 9.6×, respectively. These results confirm the necessity of a co-designed hardware-software solution for quality control.

**Controlling Quality Tradeoffs with MITHRA.** The primary goal of MITHRA is to control quality tradeoffs while preserving maximum benefits from approximate acceleration. We build an *ideal oracle design* as a gold standard to measure the efficacy of our realistic designs At any level of quality loss, the oracle *always* achieves the maximum performance and energy benefits by only filtering out the invocations that produce an accelerator error larger than the threshold. Therefore, MITHRA's objective can be redefined as a design that closely mimics the achievements of the oracle in delivering speedup and energy reduction.




**(a)** Speedup vs Application Quality Loss

**(b)** Energy Reduction vs Application Quality Loss

**(c)** Invocation Rate vs Application Quality Loss

**Figure 4.9:** We compare the mean (a) speedup, (b) energy reduction and (c) invocation rate across all the benchmarks for the oracle, table-based and neural designs for varying levels of final application output quality loss for 95% confidence interval and 90% success rate.

**Performance and energy benefits.** Figure 4.9a and Figure 4.9b shows the speedup and energy reduction when the quality tradeoffs are controlled by the oracle, the table-based design, and the neural design. These speedups and energy reductions are the *geometric mean* across all the benchmarks. We present the per-benchmark trends in Figure 4.10, and discuss them below. All the numbers in Figures 4.9b, 4.9a are presented for 90% success rate and 95% confidence interval. This result implies that with 95% confidence, we can project that at least 90% of unseen input sets will produce outputs that have quality loss level within the desired level. To obtain these results, 235 (out of 250) of the test input sets produced outputs that had the desired quality loss level. As expected, the oracle delivers the highest benefits. The table-based design and the neural design both closely follow the oracle. These results show the efficacy of both our designs in controlling the quality tradeoffs. With 5% final output quality loss, the table-based design provides 2.5× average speedup and 2.6× average energy reduction. In comparison to the table-based design, the neural design yields similar performance benefits while providing 13% more energy gains. Compared to the table-based design, the oracle achieves 26% more performance and 36% more energy benefits. Similarly, compared to the neural design, the oracle delivers 26% more speedup and 19% more energy reduction. These results suggest that both classifier designs can effectively control the tradeoff between quality and the gains from approximate acceleration.

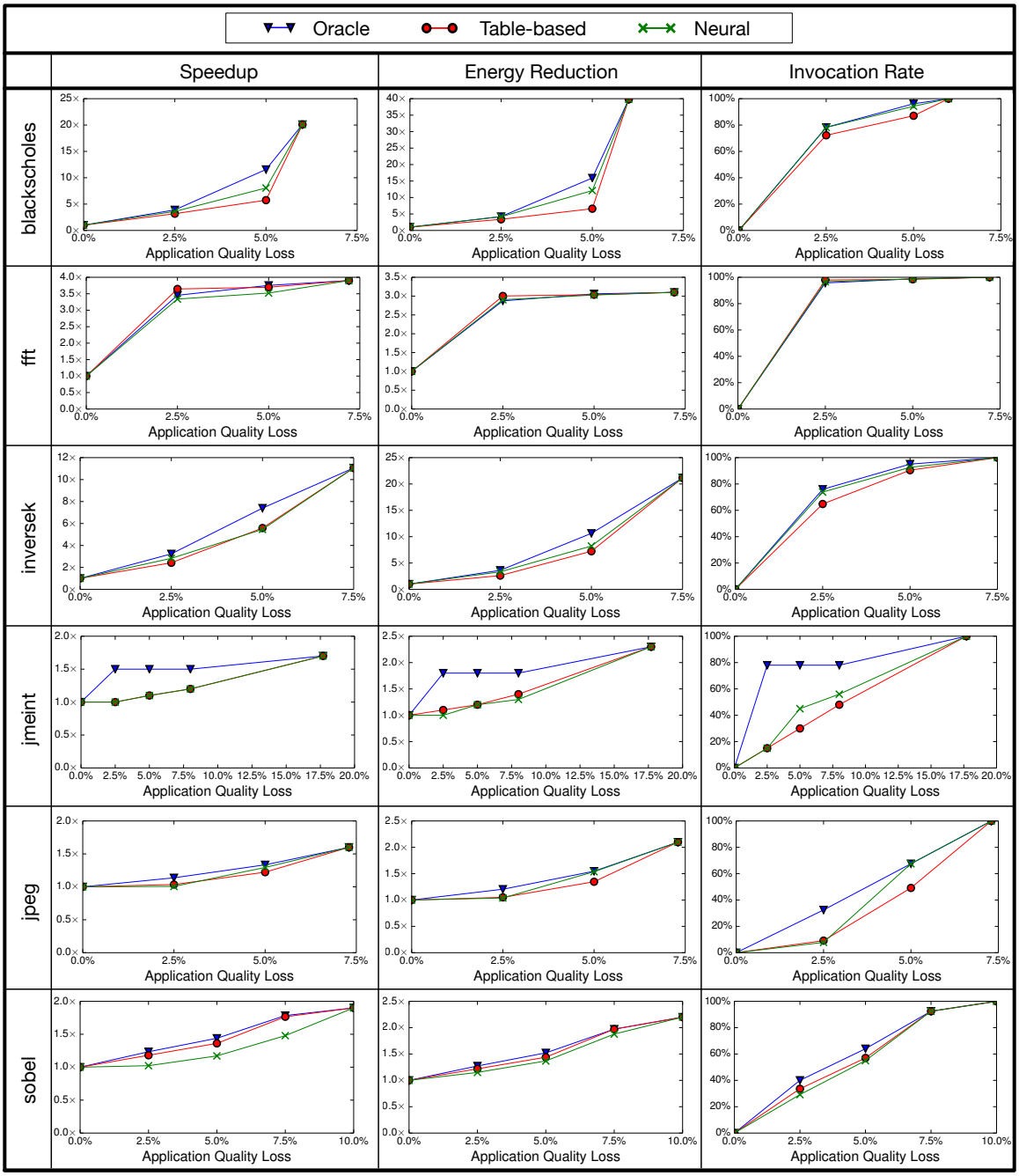



**Figure 4.10:** Speedup, energy reduction, and invocation rate for individual benchmarks at 95% confidence interval and 90% success rate.

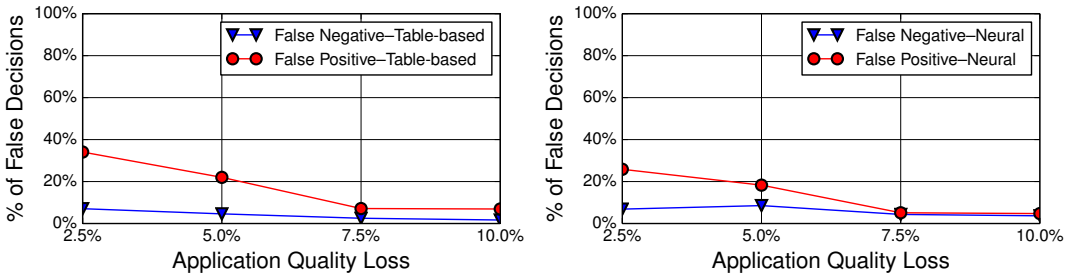
**Accelerator invocation rate.** To better understand the trends in performance and energy reduction, we examine the accelerator invocation rate with MITHRA in Figure 4.9c. The invocation rate is the percentage of target function invocations that are delegated to the accelerator. When the invocation rate is 100%, the target function is always executed on the accelerator. When the invocation rate is 0%, the function always runs on the precise core. Gains from approximate acceleration are directly proportional to invocation rate and MITHRA aims to maximize these gains for any level of quality loss. Figure 4.9c shows the invocation rate when the quality tradeoffs are controlled by the oracle, the table-based, and the neural design. As expected, the oracle provides the highest invocation rate due to its prior knowledge about all the invocations. The table-based and the neural designs obtain invocation rates that closely follow the oracle. As the quality loss level tightens, the invocation rate declines for all the three designs. For 5% quality loss level, the table-based design achieves 64% and the neural design achieves 73% average invocation rate. The oracle is only 29% and 13% higher than the table-based and neural designs, respectively. Even though the table-based design shows a lower invocation rate than the neural design, the performance achieved by both the designs are similar since the neural design generally incurs a higher cost for generating the result.

**Per-benchmark analysis.** Figure 4.10 illustrates the speedup, energy reduction, and invocation rate for each individual benchmark with 95% confidence and 90% success rate with varying desired quality levels. Similar to the mean results, the majority of benchmarks closely follow the oracle for both the designs. Two benchmarks, jmeint and jpeg, reveal interesting trends. In both cases, the neural design significantly outperforms the table-based design in terms of invocation rate. This phenomenon is the result of large number of elements in the accelerator input vector (64 inputs for jpeg and 18 inputs for jmeint). This leads to high hash conflicts and hence the table-based design is less effective in segregating inputs that incur large quality losses. Therefore, it conservatively falls back to

the original precise code to achieve better quality. Another observation is that even though jmeint achieves higher invocation rate with the neural design, the gains from approximation are similar to the table-based design. The neural design for jmeint requires a relatively large neural network with 18 input, 16 hidden, and 2 output neurons that outweighs the benefits of higher invocation rate.

**False positives and false negatives.** To further understand the operation of our classifiers, we examine their false decisions. Figure 4.11a and 4.11b shows the percentage of these false decisions (positive and negative) for the table-based and the neural design, respectively. The false positives comprise the input cases that should have been run on the accelerator according to the oracle, but are identified as potential high-error cases by classifiers and are executed using the original precise function. Conversely, the false negatives comprise those input cases, which should have been run using the original function according to the oracle but are missed by classifiers and are run on the accelerator. Figure 4.11 shows the ratio of the false decisions to the total invocations averaged over all benchmarks. With 5% quality loss, the table-based design makes 22% false positive and 5% false negative decisions. The neural design makes 18% false positive and 9% false negative decisions. The low rate of false negatives demonstrates the high efficacy of both designs in filtering out inputs that lead to large quality degradations. Moreover, the false negatives are significantly lower than the false positives with both designs because hardware classifiers adopt a conservative approach towards controlling quality tradeoffs; hence, prioritizing quality over benefits from approximation.

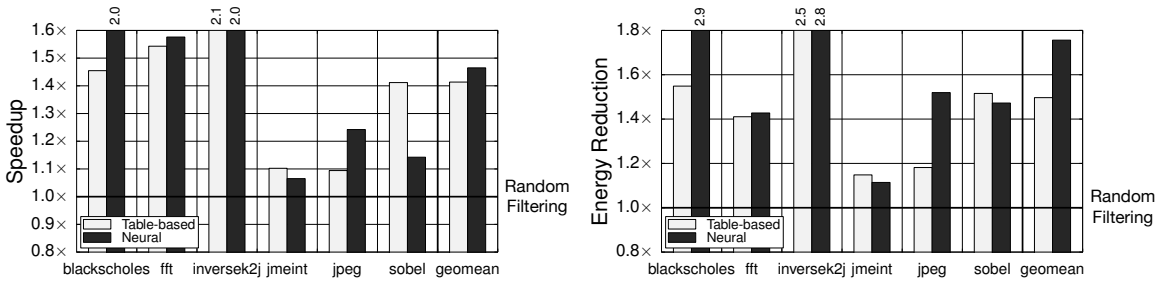
**Comparison with random filtering.** We compare our input-conscious techniques to a simple random filtering technique. In this technique, the decision to delegate a function invocation to the accelerator is random, irrespective of the inputs. Figure 4.12 shows the speedup and energy reduction with both of our techniques relative to the random filtering at 5% quality loss. The trends are similar for other quality levels. Compared to random




**(a)** False Positives and False Negatives (Table Design)

**(b)** False Positives and False Negatives (Neural Design)

**Figure 4.11:** False positive and false negative decisions for (a) table-based and (b) neural classifier for varying quality losses at 95% confidence interval and 90% success rate.




**(a)** Speedup with MITHRA (Baseline: Random Filtering)

**(b)** Energy Reduction with MITHRA (Baseline: Random Filtering)

**Figure 4.12:** (a) Speedup and (b) energy reduction for 95% confidence interval and 90% success rate compared to random filtering at 5% quality loss. The baseline is approximate acceleration with random filtering.

filtering, the table-based design delivers 41% average speedup and 50% average energy reduction. With the neural design, these figures increase to 46% average speedup and 76% average energy reduction. The speedup is as high as 2.1× (inversek2j with the table-based design) and the maximum energy reduction grows to 2.9× reduction (blackscholes with the neural design). These results collectively confirm the importance of focusing and capturing the inputs that lead to large quality losses.

**Varying success rate with 95% confidence.** MITHRA aims to provide statistical guarantees with high confidence that quality levels will be met. The main results focus on a confidence interval of 95% and attain 90% success rate. As the Clopper-Pearson method in Section 4.4.2 describes, for different confidence intervals the success rate is dependent

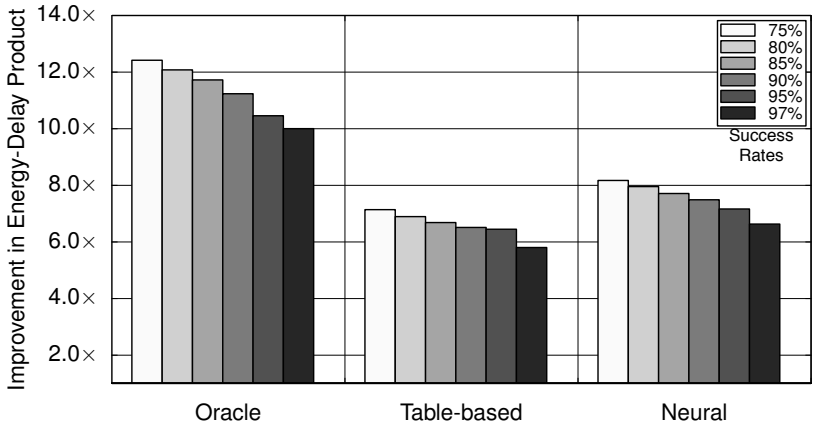



**Figure 4.13:** Trends in the Energy-Delay product for varying success rate with 95% confidence interval and at 5% quality loss level.

on $n_{success}$ which in turn is dependent on the threshold selected. For 5% quality loss level, we vary the threshold so as to obtain a sweep of success rates for 95% confidence interval. Figure 4.13 presents the improvement in energy-delay product for these success rates. As the success rate increases, implying that there is a higher probability that quality levels will be met, the benefits from approximation decrease. Higher success rate provides higher statistical guarantee and therefore comes at a higher price. The results show the MITHRA effectively enables the programmer to control both the level of quality loss and also the degree of statistical guarantee they desire.

**Pareto Analysis for the Table-Based Classifier** The two main design parameters of the table-based design are the number of parallel tables and the size of each table. We vary these two parameters to explore the design space of table-based MITHRA and perform Pareto analysis to find the optimal configuration. Figure 4.14 illustrates the Pareto analysis for 5% quality loss. The trends are similar with other levels of quality loss. The design that is denoted by ($a$T×$b$KB) represents a configuration with $a$ parallel tables, each of size size $b$ kilo bytes. We explore 16 different designs, a set of all the combinations of (1T, 2T, 4T, 8T) parallel tables and (0.125KB, 0.5KB, 2KB, 4KB) table sizes. The $x$-axis in Figure 4.14 captures uncompressed size in kilo bytes. The $y$-axis is the average accelerator invocation rate across all benchmarks. We use the average invocation rate because the invocation

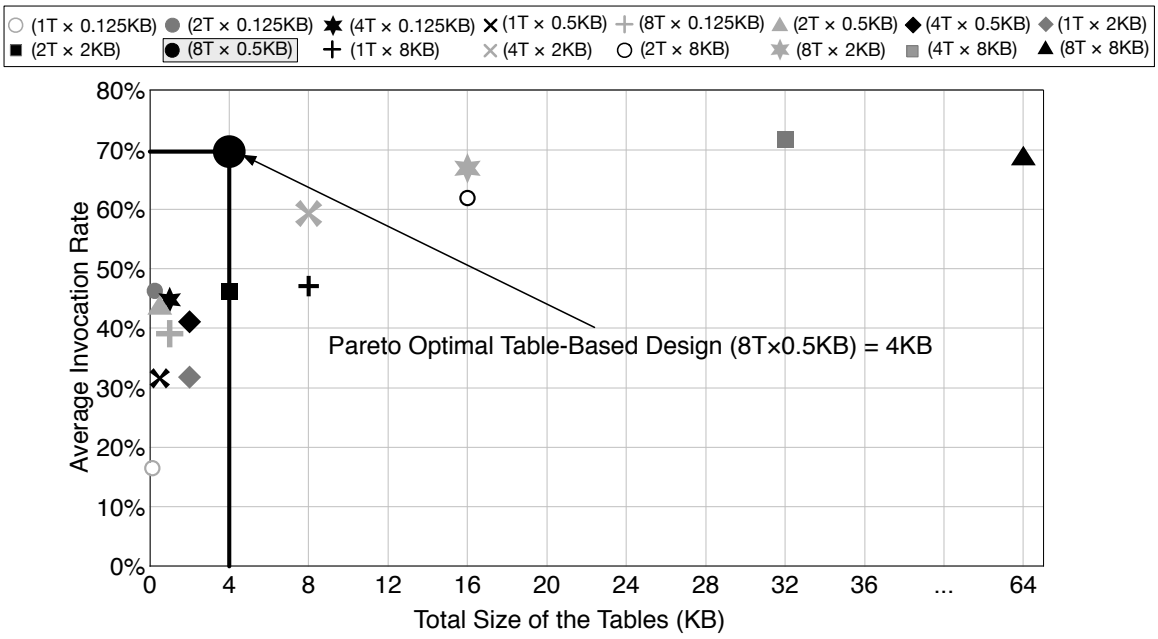



**Figure 4.14:** Pareto analysis for the table-based MITHRA at 5% quality loss. The ($a$T×$b$KB) is a configuration with $a$ parallel tables each of size $b$ KB. Our default configuration, (8T×0.5KB), is Pareto optimal.

rate directly determines the speedup and efficiency benefits. In Figure 4.14, the optimal design minimizes the size of the predictor (left on the $x$-axis) and maximizes the accelerator invocation rate (up on the $y$-axis). As Figure 4.14 illustrates, the design with eight parallel tables, each of size 0.5KB is the Pareto optimal design. This design space exploration shows that both the number of tables and the size of each table have a significant impact on the accelerator invocation rate. Due to destructive aliasing, a smaller table (0.125KB) is unable to discriminate the inputs that cause large errors. As a result, smaller tables fail to preserve the benefit of acceleration. On the other hand, larger table sizes (8KB) do not provide benefits beyond a certain point because destructive aliasing cannot be completely eliminated even with 8KB tables. Hence, we use the Pareto optimal point with 8 tables each of size 0.5KB as our default configuration. The configuration with larger number of tables provide higher benefits even if the size of each table is small as the chance of destructive aliasing decreases since each table uses a distinct hash function.

**Data Compression for the Table-based MITHRA** Based on the Pareto analysis, the optimal table-based design is eight tables, each of size 0.5 KB. Therefore, the total uncompressed size of this design is 4 KB. We observe that there are large trails of 0s in the tables. This insight provides an opportunity to compress the table and reduce the necessary memory state of the table-based design. To compress the tables, we use the low-overhead and low-latency Base-Delta compression technique [159] that has been recently proposed for cache line compression. The Base-Delta compression and decompression algorithms require only vector addition, subtraction, and comparison operations. We arrange the tables in rows of 64 B size to employ this cache line compression mechanism. Table 4.3 shows the compression results for each benchmark from the original uncompressed size of 4 KB. The sparser the contents of the table, the higher the compression ratio. These results show that blackscholes, fft, inversek2j, and jmeint achieve 16× size reduction. However, sobel and jpeg do not benefit from compression due to the complexity of the inputs and the high density of the contents of the tables. Table 4.3 also shows the size of the neural MITHRA for each application and their topology. In most cases, after compression, the sizes of MITHRA is less than 1 KB.

### 4.4.7 Related Work

A growing body of work has explored leveraging approximation for gains in performance and energy [139, 134, 140, 141, 147, 150, 148, 142, 141, 143, 153, 72]. Our work, however, focuses on a hardware-software mechanism to control quality tradeoffs for approximate accelerators. Several techniques provide software-only quality control mechanisms for approximate computing that either operate at compile-time [144, 145, 135, 164, 165] or runtime [143, 141, 142, 166, 153]. In contrast, we define MITHRA, that uses hardware to control quality tradeoffs at runtime and provides necessary compiler support for the proposed hardware designs. Below, we discuss the most related works.

**Compile-time techniques to control quality tradeoffs.** Rely [135] is an approximate programming language that requires programmers to mark variables and operations as approximate. Given these annotations, Rely combines symbolic and probabilistic reasoning to verify whether the quality requirements are satisfied for a function. To provide this guarantee, Rely requires the programmer to not only mark all variables and operations as approximate but also provide preconditions on the reliability and range of the data. Similar to Rely, the work in [165] proposes a relational Hoare-like logic to reason about the correctness of approximate programs. The work in [164] uses Bayesian network representation and symbolic execution to verify probabilistic assertions on the quality of the approximate programs given the input distributions. Given a quality requirement, Chisel [145] uses Integer Linear Programming (ILP) to optimize the approximate computational kernels at compile time. The approximation model in these works is based on architectures that support approximation at the granularity of a single instruction [140]. The work in Stoke [167] focuses on reducing the bit width of floating-point operations at compile-time, trading accuracy for performance. While these techniques focus approximation at the fine granularity of single instruction, we focus on coarse-gain approximation with accelerators. In a concurrent work [168], determining the quality control knob is cast as an optimization problem; however, the work neither uses statistical techniques nor provides hardware mechanisms for quality control. The above approaches do not utilize runtime information or propose microarchitectural mechanisms for controlling quality tradeoffs, which is a focus of our work.

**Runtime quality control techniques.** Green [143] provides a code-centric programming model for annotating loops for early termination and substitution of numerical functions with approximate variants. Sage [141] and Paraprox [142] provide a set of static approximation techniques for GPGPU kernels. Both techniques utilize the CPU to occasionally monitor the quality of the final outputs and adjust the approximation level. Approx-

Hadoop [169] uses statistical sampling theory to control input sampling and task dropping in approximate MapReduce tasks. Light-Weight Checks [166] requires the programmer to write software checks for each approximately accelerated function and the precise function is run if the check fails. Rumba [170], concurrent to an earlier version of this work [171], only proposes microarchitectural mechanisms that use decision trees and linear models for predicting the accelerator error value. Since Rumba does not offer the necessary compiler support, it does not map the final output quality to the local decision on the accelerator call site. The lack of compiler support impedes Rumba from providing concrete statistical guarantees for the final output quality. Rumba also relies on error value prediction (regression) that is significantly more demanding and less reliable than the MITHRA's binary classification solution.

Unlike these techniques that either rely only hardware or software checks, we define a cohesively co-designed hardware-software technique for controlling quality tradeoffs that leverages runtime information and compiler support to maximize the gains from approximate acceleration and provide statistical guarantees.

### 4.4.8 Conclusion

Approximate accelerators deliver significant gains in performance and efficiency by trading small losses in quality of the results. However, the lack of a hardware-software mechanism that control this tradeoff limit their applicability. In this paper, we describe MITHRA, a hardware-software solution for controlling the quality tradeoffs at runtime. MITHRA provides only statistical quality guarantees on unseen data. The acceptability of such guarantees is still a matter of debate and investigation. However, it is clear that the applicability of approximate computing requires moving beyond traditional and formal quality guarantees. In fact, such guarantees are to some extent accepted for service level agreements in data centers. The widely used machine learning algorithms also rely on similar statistical guarantees. This work takes an initial step in controlling the quality tradeoffs for approximate

accelerators; aiming to open a path for their adoption.

# CHAPTER 5
# OTHER WORKS BY THIS AUTHOR

TABLA was the inception of the broader artificial intelligence effort in the Alternative Computing Technologies Lab led by Professor Hadi Esmaeilzadeh. Building on the concept of template-based architectures that can be customized for the application by a specialized full stack solution, we developed CoSMIC, a natural extension of TABLA, which accelerates classical machine learning at scale [86] and is described in Section 5.1. It is a pioneering effort in bridging the gap between distributed systems and accelerators by developing a specialized full stack for scale-out acceleration of machine learning. In another concurrent feat, we developed, DNNWEAVER, a template-based approach to accelerate the inference phase of Deep Neural Networks [94] (Section 5.2). Taking the idea of leveraging the common properties of an application domain, we devised *RoboX* [172], discussed in Section 5.3. *RoboX* is based on the concept that many motion planning and control algorithms can be formulated as a constrained optimization problem solved through Model Predictive Control (MPC). Finally, my colleagues and I devised AXGAMES, described in Section 5.4, that aims to understand the user perspective on approximate results, such as images, audio, video and other application outputs which are easily discernible. Navigating the tradeoffs to determine the acceptable level of quality is challenging for developers, and affects the degree of approximation or whether approximation is used at all. AXGAMES aims to quantify the acceptable levels of quality loss for users to assist the developers to employ approximation techniques that are within this boundary.

## 5.1 Scale-Out Acceleration for Machine Learning

TABLA and DAnA target single node acceleration, i.e., single FPGA hardware accelerators, however, the growing scale and complexity of machine learning algorithms has resulted in prevalent use of distributed general-purpose systems. In a rather disjoint effort, the community is focusing mostly on high performance single-node accelerators for learning. This work bridges these two paradigms and offers CoSMIC, a full computing stack constituting language, compiler, system software, template architecture, and circuit generators, that enable programmable acceleration of learning at scale. CoSMIC enables programmers to exploit scale-out acceleration using FPGAs and Programmable ASICs (P-ASICs) from a high-level and mathematical Domain-Specific Language (DSL). Nonetheless, CoSMIC does not require programmers to delve into the onerous task of system software development or hardware design. CoSMIC achieves three conflicting objectives of efficiency, automation, and programmability, by integrating a novel multi-threaded template accelerator architecture and a cohesive stack that generates the hardware and software code from its high-level DSL. CoSMIC can accelerate a wide range of learning algorithms that are most commonly trained using parallel variants of gradient descent. The key is to distribute partial gradient calculations of the learning algorithms across the accelerator-augmented nodes of the scale-out system. Additionally, CoSMIC leverages the parallelizability of the algorithms to offer multi-threaded acceleration within each node. Multi-threading allows CoSMIC to efficiently exploit the numerous resources that are becoming available on modern FPGAs/P-ASICs by striking a balance between multi-threaded parallelism and single-threaded performance. CoSMIC takes advantage of algorithmic properties of machine learning to offer a specialized system software that optimizes task allocation, role-assignment, thread management, and internode communication. We evaluate the versatility and efficiency of CoSMIC for 10 different machine learning applications from various domains. On average, a 16-node CoSMIC with UltraScale+ FPGAs offers 18.8× speedup

over a 16-node Spark system with Xeon processors, while the programmer only writes 22–55 lines of code. CoSMIC offers higher scalability compared to the state-of-the-art Spark. Scaling from 4 to 16 nodes with CoSMIC yields 2.7× improvements whereas Spark offers 1.8×. These results confirm that the full-stack approach of CoSMIC takes an effective and vital step towards enabling scale-out acceleration for machine learning.

## 5.2 From High-Level Deep Neural Models to FPGAs

Deep Neural Networks (DNNs) are compute-intensive learning models with growing applicability in a wide range of domains. FPGAs are an attractive choice for DNNs since they offer a programmable substrate for acceleration and are becoming available across different market segments. However, obtaining both performance and energy efficiency with FPGAs is a laborious task even for expert hardware designers. Furthermore, the large memory footprint of DNNs, coupled with the FPGAs' limited on-chip storage makes DNN acceleration using FPGAs more challenging. This work tackles these challenges by devising DNNWEAVER, a framework that *automatically generates* a synthesizable accelerator for a given {DNN, FPGA} pair from a high-level specification in Caffe [20]. To achieve large benefits while preserving automation, DNNWEAVER generates accelerators using hand-optimized design templates. First, DNNWEAVER translates a given high-level DNN specification to its novel ISA that represents a macro dataflow graph of the DNN. The DNNWEAVER compiler is equipped with our optimization algorithm that tiles, schedules, and batches DNN operations to maximize data reuse and best utilize target FPGA's resources. The final result is a custom synthesizable accelerator that best matches the needs of the DNN while providing high performance and efficiency gains for the target FPGA.

We use DNNWEAVER to generate accelerators for a set of eight different DNN models and three different FPGAs ( Xilinx Zynq, Altera Stratix V, and Altera Arria 10). We use hardware measurements to compare the generated accelerators to both multicore CPUs (ARM Cortex A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650Ti, and Tesla K40). In

comparison, the generated accelerators deliver superior performance and efficiency without requiring the programmers to participate in the arduous task of hardware design.

## 5.3 Hardware Acceleration for Autonomous Control in Robotics

TABLA and DAnA have targeted classical machine learning training as their target application domain due to its intense compute requirements. In addition to such algorithms, there is an emerging opportunity to use programmable hardware accelerators as an alternative to general-purpose platforms to meet the growing compute demands of intelligent robotic systems. Novel algorithmic advances have paved the way for robotics to transform the dynamics of many social and enterprise applications. Even though true autonomy in robotics requires sophisticated perception algorithms, it hinges upon compute-intensive motion planning and control algorithms which allow the robot to continuously process and interact with an uncertain and dynamic environment under constrained power budgets. Specialized architectures offer a potent choice to provide low-power, high-performance accelerators to address these unique challenges and deliver greater compute capabilities. Instead of taking a traditional route which profiles and maps hot code regions to accelerators, this work delves into the algorithmic characteristics of the application domain. A key insight of our next work, *RoboX* [172], is that many motion planning and control algorithms are formulated as a constrained optimization problem solved online through Model Predictive Control (MPC). While models and objective functions differ between robotic systems and tasks, the structure of the optimization problem and solver remain fixed. Therefore, instead of simply designing a hardware accelerator, we develop an end-to-end acceleration compute stack which exposes a novel domain specific language close to the mathematical descriptions used by roboticists. This interface allows roboticists to concisely express the physics of the robot and its task in a form close to its mathematical expressions. The *RoboX* backend then automatically maps this high-level specification to a novel programmable architecture, which harbors a programmable memory access engine and compute-enabled

interconnects. Hops in the interconnect are augmented with simple functional units that either operate on in-fight data or are bypassed according to a micro-program. Evaluations with six different robotic systems and tasks show that *RoboX* provides a 29.4× (7.3 ×) speedup and 22.1× (79.4×) performance-per-watt improvement over an ARM Cortex A57 (Intel Xeon E3). Compared to GPUs, *RoboX* attains 7.8×, 65.5×, and 71.8× higher Performance-per-Watt to Tegra X2, GTX 650 Ti, and Tesla K40, respectively, with a power envelope of only 3.4 Watts at 45 nm.

## 5.4 Towards Crowdsourcing Quality Target Determination in Approximate Computing

Approximate computing trades quality of application output for higher efficiency and performance. Approximation is useful only if its impact on application output quality is acceptable to the users. However, there is a lack of systematic solutions and studies that explore users' perspective on the effects of approximation. In this paper, we seek to provide one such solution for the developers to probe and discover the boundary of quality loss that most users will deem acceptable. We propose AxGames, a crowdsourced solution that enables developers to readily infer a statistical common ground from the general public through three entertaining games. The users engage in these games by betting on their opinion about the quality loss of the final output while the AxGames framework collects statistics about their perceptions. The framework then statistically analyzes the results to determine the acceptable levels of quality for a pair of (application, approximation technique). The three games are designed such that they effectively capture quality requirements with various tradeoffs and contexts.

To evaluate AxGames, we examine seven diverse applications that produce user perceptible outputs and cover a wide range of domains, including image processing, optical character recognition, speech to text conversion, and audio processing. We recruit 700

participants/users through Amazon's Mechanical Turk to play the games that collect statistics about users perception on different levels of quality. Subsequently, the AxGames framework uses the Clopper-Pearson exact method, which computes a binomial proportion confidence interval, to analyze the collected statistics for each level of quality. Using this analysis, AxGames can statistically project the quality level that satisfies a given percentage of users. The developers can use these statistical projections to tune the level of approximation based on the users' experience. We find that the level of acceptable quality loss significantly varies across applications. For instance, to satisfy 90% of the users, the level of acceptable quality loss is 2% for one application (image processing) and 26% for another (audio processing). Moreover, the pattern with which the crowd responds to approximation takes significantly different shapes and forms depending on the class of applications. These results confirm the necessity of solutions that systematically explore the effect of approximation on the end user experience.

# CHAPTER 6

# LOOKING FORWARD

Even though the end of Moore's law poses a huge challenge, it enables our community to innovate at the intersection of multiple disciplines. It is evident from the complexity of problems we are currently seeing that performing isolated research is no longer a viable option. Consequently, the below mentioned future research directions aim to devise faster and more energy efficient compute systems at the cross-section of: (1) machine learning and robotics planning and control algorithms, (2) data collection and management systems, and (3) sustainability and technology for emerging economies. In this thesis, we focused on machine learning for data analytics and its interplay with row-based relational database management systems. Figure 6.1 encapsulates some of the future directions which are unexplored in the data processing pipeline. As we move forward, the following are some of the promising research directions.

## 6.1 Solutions for Server-Scale Data Processing Pipeline

The goal here is to explore the properties of the entire horizontal pipeline of data processing from data collection and management to the final data analysis. This would require understanding the characteristics of the flow of data in the pipeline in order to design server-scale hardware acceleration frameworks that integrate within these complex application domains. Previously we have looked at the algorithmic properties of classical machine learning, however, numerous other algorithms such as Bayesian networks, Markov chains, graph-powered learning, self-organizing maps, topological data analysis, and Gaussian processes, exhibit significantly different compute patterns, hence require us to rethink the accelerations frameworks. Additionally, complex data management systems are gain-

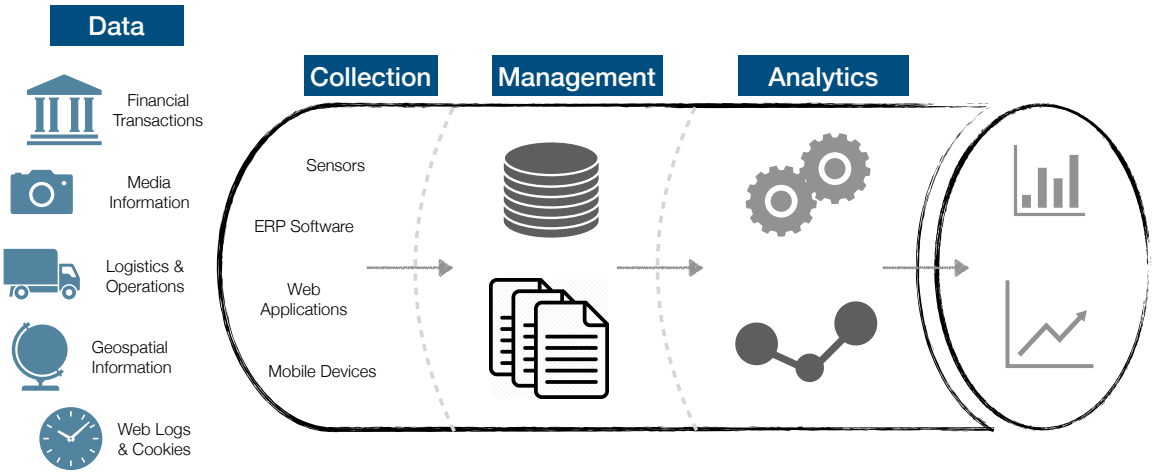



**Figure 6.1:** A visualization of the data processing pipeline. We have only touched a part of this pipeline by integrating FPGA-based hardware accelerators for machine learning within Relational Database Management Systems. As we touch this tip of the iceberg, we observe numerous other challenges that are waiting to be tackled.

ing traction as data is being generated with high velocity, veracity, dimensionality, and does not necessarily have a structure. Future work can focus on non-conventional databases such as column-based and graph-based databases, which exhibit different challenges than row-based relational database management systems (a focus of DAnA). Challenges for column-based databases include extracting features for a machine learning algorithm that are spread over different columns and stored across memory. Similarly graph-based databases require systems that are designed to tackle irregular memory access. Furthermore, web applications generate unstructured and semi-structured data and converting such formats to adhere to relational database systems would require unnecessary processing of large amounts of data. Instead, one future direction is to devise systems that directly process data-JSON and XML structures and extract relevant features in accordance to the query instead of parsing the entire information.

## 6.2 Pushing Intelligence to the Edge

In this direction, the goal would be to design real-time compute systems for Internet of Things (IoT) and edge devices that can run machine learning applications whilst provid-

ing light-weight support for managing the data collected from their sensors. Such devices operate under constrained power budgets, hence require content-aware solutions that efficiently manage and process only the required data. Furthermore, the data collected from these devices is often noisy, corrupted, and redundant. Therefore, the plan would be to design on-edge reconfigurable hardware systems that can perform intelligent data filtering, interpolation, sampling and processing. Moreover, these devices also generate and manipulate time-series data. Thus, another future direction is to investigate the algorithms that examine temporal correlations with a goal to perform design exploration of systems that are adapted for such applications. This research direction aims to understand the challenges in data collection, management, and preliminary data analysis on edge devices.

## 6.3 Architectural Support for Geospatial Analytics

As mobile devices and IoTs become ubiquitous, they are generating data that have an intrinsic geographical dimension, i.e., there is a location associated with this information. Even though there are established software solutions specialized for analyzing topographical information, there is a lack of hardware designs that are tailored towards such data structures and algorithms. With my experience in algorithmic analysis, I will characterize these workloads to identify the constraints and bottlenecks of the current systems. Post analysis, my goal will be to investigate algorithms and databases that are specialized to perform analytics on geographic objects. The focus will be to design systems that can furnish capabilities such as managing and processing raster and vector data to execute high performance spatial queries and complex geospatial analytics.

## 6.4 Emerging Technologies in Developing Economies

As a computer architect specializing in hardware design from Georgia Institute of Technology, I am aware of how United States is the powerhouse for technology. However, having

traveled to several other countries, I have gained a global perspective and seen first-hand the big gap in technological advancements. People are striving to bridge this gap and improve the living conditions of those in developing nations, who do not have the privilege and access to modern developments, such as fast internet and smart devices, which we take for granted. Therefore, the challenges in these economies need to be tackled differently from our traditional approaches. The system design needs to account for the available IT resources, with the goal to boost data collection, storage, management, and analysis centered around the cultural norms and grassroots economics.

Many fields in computer science are evolving rapidly; computer architecture is following a similar if not a steeper trend. This can be attributed to the fact that the current paradigm of general-purpose processor design falls significantly short of the traditional cadence of performance improvements. Nonetheless, it brings forth an opportunity to find unique ways to satisfy the societal demand of high performance compute. In these times of rediscovery, collaboration across multiple disciplines is inevitable, and already in motion. I believe in developing interdisciplinary solutions and breakthrough technologies that tackle challenges across different domains in computer science.

## REFERENCES


[1] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[3] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, 1974.

[4] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, Jun. 2014.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017. arXiv: `1704.04760`.

[6] *Amazon EC2 F1 instances: Run custom FPGAs in the amazon web services (aws) cloud*, `https://aws.amazon.com/ec2/instance-types/f1/`, 2017.

[7] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, 2011.

[8] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS*, 2010.

[9] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *MIRO*, ser. MICRO-44, 2011.

[10] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[11] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *ASPLOS*, 2015.

[12] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. Kim, and Hadi Esmaeilzadeh, "TABLA: A UNIFIED TEMPLATE-BASED FRAMEWORK FOR ACCELERATING STATISTICAL MACHINE LEARNING," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016.

[13] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-rdbms hardware acceleration of advanced analytics," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1317–1331, Jul. 2018.

[14] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, Hadi Esmaeilzadeh, and K. Bazargan, "Axilog: Language support for approximate hardware design," in *ACM/IEEE Conference on Design Automation and Test in Europe (DATE)*, Mar. 2015, pp. 812–817.

[15] D. Mahajan, K. Ramkrishnan, R. Jariwala, A. Yazdanbakhsh, J. Park, B. Thwaites, A. Nagendrakumar, A. Rahimi, Hadi Esmaeilzadeh, and K. Bazargan., "Axilog: Abstractions for approximate hardware design and reuse," *IEEE Micro special issue on Alternative Computing Designs and Technologies*, vol. 35, no. 5, pp. 16–30, 2015.

[16] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and Hadi Esmaeilzadeh, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2016.

[17] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "CPU DB: Recording microprocessor history," *ACM Queue*, vol. 10, no. 4, 10:10–10:27, Apr. 2012.

[18] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[19] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-RDBMS analytics," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '12, 2012.

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[21] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On parallelizability of stochastic gradient descent for speech dnns," in *ICASSP*, 2014.

[22] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Neural Information Processing Systems*, 2010.

[23] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, no. Jan, pp. 165–202, 2012.

[24] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.

[25] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. D. Walker, "Efficient large-scale distributed training of conditional maximum entropy models," in *NIPS*, 2009.

[26] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," *arXiv:1602.06709 [cs]*, 2016.

[27] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *International Conference on Learning Representations Workshop Track*, 2016.

[28] *Amazon web services postgresql*, `https://aws.amazon.com/rds/postgresql/`.

[29] *Azure sql database*, `https://azure.microsoft.com/en-us/services/sql-database/`.

[30] *Gartner Report on Analytics*, `gartner.com/it/page.jsp?id=1971516`.

[31] *SAS Report on Analytics*, `sas.com/reg/wp/corp/23876`.

[32] *Oracle Data Mining*, `http://www.oracle.com/technetwork/database/options/advanced-analytics/odm/overview/index.html`.

[33] *Oracle R Enterprise*, http://www.oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/index.html.

[34] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, "MAD Skills: New Analysis Practices for Big Data," *PVLDB*, vol. 2, no. 2, pp. 1481–1492, 2009.

[35] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The MADlib Analytics Library: Or MAD Skills, the SQL," *PVLDB*, vol. 5, no. 12, pp. 1700–1711, 2012.

[36] *Microsoft SQL Server Data Mining*, https://docs.microsoft.com/en-us/sql/analysis-services/data-mining/data-mining-ssas.

[37] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2011.

[38] Y. Cheng, C. Qin, and F. Rusu, "GLADE: Big Data Analytics Made Easy," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA: ACM, 2012, pp. 697–700, ISBN: 978-1-4503-1247-9.

[39] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: ACM, 2015, pp. 1969–1984, ISBN: 978-1-4503-2758-9.

[40] S. Sirowy and A. Forin, "Where's the beef? why FPGAs are so fast," Microsoft Research, Tech. Rep. MSR-TR-2008-130, Sep. 2008.

[41] Xilinx, *Zynq-7000 all programmable soc*, 2014.

[42] Intel Corporation, *Disrupting the data center to create the digital services economy*.

[43] D. C. Ku and G. De Micheli, *High level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic Publishers, 1992.

[44] M. Lichman, *UCI machine learning repository*, 2013.

[45] I. Cantador, P. Brusilovsky, and T. Kuflik, "Second workshop on information heterogeneity and fusion in recommender systems (HetRec)," in *Proceedings of the ACM conference on Recommender systems*, ser. RecSys 2011, 2011.

[46] Grouplens. (). Movielens dataset.

[47] K. A. Grajski, "Neurocomputing, using the MasPar MP-1," in *Parallel Digital Implementations of Neural Networks*, K. W. Przytula and V. K. Prasnna, Eds., Prentice-Hall, 1993, ch. 2, pp. 51–76.

[48] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[49] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[50] Nvidia, *Jetson*, `http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html`, 2015.

[51] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, Jun. 2008.

[52] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[53] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on loongson 3A processor," in *ICPADS*, 2012.

[54] S. Rendle, "Factorization machines with libFM," *ACM Trans. Intell. Syst. Technol.*, vol. 3, no. 3, 57:1–57:22, May 2012.

[55] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, 27:1–27:27, May 2011.

[56] S. Nissen, "Implementation of a fast artificial neural network library (FANN)," Department of Computer Science University of Copenhagen (DIKU), Tech. Rep., 2003, http://fann.sf.net.

[57] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: A polyvalent machine learning accelerator," in *ASPLOS*, 2015.

[58] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for support vector machines," in *12th International Workshop on Image Analysis for Multimedia Interactive Services*, 2011.

[59] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient primitives for deep learning," *CoRR*, 2014.

[60] I. Stamoulias and E. S. Manolakos, "Parallel architectures for the knn classifier – design of soft IP cores and FPGA implementations," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, 22:1–22:21, Sep. 2013.

[61] E. Manolakos and I. Stamoulias, "IP-cores design for the knn classifier," in *ISCAS*, 2010.

[62] H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data," in *AHS*, 2011.

[63] T. Maruyama, "Real-time K-Means clustering for color images on reconfigurable hardware," in *ICPR*, 2006, pp. 816–819.

[64] A. Filho, A. Frery, C. de Araujo, H. Alice, J. Cerqueira, J. Loureiro, M. de Lima, M. Oliveira, and M. Horta, "Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm," in *SBCCI*, 2003.

[65] M. Papadonikolakis and C. Bouganis, "A heterogeneous FPGA architecture for support vector machine training," in *FCCM*, 2010.

[66] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. Graf, "A massively parallel fpga-based coprocessor for support vector machines," in *FCCM*, 2009.

[67] A. Majumdar, S. Cadambi, and S. Chakradhar, "An energy-efficient heterogeneous system for embedded learning and classification," *Embedded Systems Letters, IEEE*, vol. 3, no. 1, pp. 42–45, 2011.

[68] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf, "A massively parallel, energy efficient programmable accelerator for learning and classification," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, 6:1–6:30, Mar. 2012.

[69] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, 2011, pp. 109–116.

[70] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[71] A. Maashri, M. DeBole, M. Cotter, N. Chandramoorthy, Y. Xiao, V. Narayanan, and C. Chakrabarti, "Accelerating neuromorphic vision algorithms for recognition," in *DAC*, 2012.

[72] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable socs via neural acceleration," in *HPCA*, 2015.

[73] D. Kesler, B. Deka, and R. Kumar, "A hardware acceleration technique for gradient descent and conjugate gradient," in *SASP*, 2011.

[74] A. Roldao and G. A. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 1, 1:1–1:19, Jan. 2010.

[75] G. Morris, V. Prasanna, and R. Anderson, "A hybrid approach for mapping conjugate gradient onto an fpga-augmented reconfigurable supercomputer," in *FCCM*, 2006.

[76] D. DuBois, A. DuBois, T. Boorman, C. Connor, and S. Poole, "An implementation of the conjugate gradient algorithm on fpgas," in *FCCM*, 2008.

[77] Y.-J. Yeh, H.-Y. Li, W.-J. Hwang, and C.-Y. Fang, "FPGA implementation of kNN classifier based on wavelet transform and partial distance search," in *SCIA*, 2007.

[78] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures," in *FPGA*, 2008.

[79] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *FPGA*, 2010.

[80] J. Fowers, K. Ovtcharov, K. Strauss, E. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *FCCM*, IEEE, 2014.

[81] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for fpga-based computing," in *FPGA*, 2011.

[82] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big data on little clients," in *ISCA*, 2013.

[83] M. King, A. Khan, A. Agarwal, O. Arcas, and Arvind, "Generating infrastructure for FPGA-accelerated applications," in *FPL*, 2013.

[84] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid cpu-fpga databases," in *2017 IEEE 25th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 211–218.

[85] D. Sidler, M. Owaida, Z. István, K. Kara, and G. Alonso, "Doppiodb: A hardware accelerated database," in *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, 2017, p. 1.

[86] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, Massachusetts: ACM, 2017, pp. 367–381, ISBN: 978-1-4503-4952-9.

[87] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, IEEE, 2014, pp. 609–622.

[88] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics," *PVLDB*, vol. 9, no. 14, pp. 1647–1658, 2016.

[89] R. Mueller, J. Teubner, and G. Alonso, "Data processing on fpgas," *PVLDB*, vol. 2, no. 1, pp. 910–921, 2009.

[90] K. Kara, J. Giceva, and G. Alonso, "Fpga-based data partitioning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: ACM, 2017, pp. 433–445, ISBN: 978-1-4503-4197-4.

[91] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "Fpga-accelerated dense linear machine learning: A precision-convergence trade-off," *2017 IEEE 25th FCCM*, pp. 160–167, 2017.

[92] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, pp. 1–13.

[93] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *HPCA*, 2016.

[94] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and Hadi Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Oct. 2016.

[95] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "Strads: A distributed framework for scheduled model parallel machine learning," in *Pro-*

*ceedings of the 11th European Conference on Computer Systems*, London, United Kingdom: ACM, 2016, 5:1–5:16, ISBN: 978-1-4503-4240-7.

[96] C. Zhang and C. Ré, "Dimmwitted: A study of main-memory statistical analytics," *Computing Research Repository (CoRR)*, vol. abs/1403.7550, 2014. arXiv: 1403.7550.

[97] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A massively parallel fpga-based coprocessor for support vector machines," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 115–122.

[98] M. Papadonikolakis and C. S. Bouganis, "A heterogeneous fpga architecture for support vector machine training," in *2010 18th IEEE FCCM*, 2010, pp. 211–214.

[99] *Falcon computing*, http://cadlab.cs.ucla.edu/~cong/slides/HALO15_keynote.pdf.

[100] (). TABLA source code. http://www.act-lab.org/artifacts/tabla/, (visited on 03/10/2018).

[101] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "Monetdb: Two decades of research in column-oriented database architectures," *IEEE Technical Committee on Data Engineering*, vol. 35, no. 1, pp. 40–45, 2012.

[102] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[103] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, Seoul, Republic of Korea: IEEE Press, 2016, pp. 243–254, ISBN: 978-1-4673-8947-1.

[104] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.

[105] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *ISCA*, 2016.

[106] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.

[107] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[108] B. L. Milenova, J. S. Yarmus, and M. M. Campos, "Svm in oracle database 10 g : Removing the barriers to widespread adoption of support vector machines," *PVLDB*, pp. 1152–1163, 2005.

[109] D. Z. Wang, M. J. Franklin, M. Garofalakis, and J. M. Hellerstein, "Querying probabilistic information extraction," *PVLDB*, vol. 3, no. 1-2, pp. 1057–1067, 2010.

[110] M. Wick, A. McCallum, and G. Miklau, "Scalable probabilistic databases with factor graphs and mcmc," *PVLDB*, vol. 3, no. 1-2, pp. 794–804, 2010.

[111] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: ACM, 2013, pp. 13–24, ISBN: 978-1-4503-2037-5.

[112] M. L. Koc and C. Ré, "Incrementally maintaining classification using an rdbms," *PVLDB*, vol. 4, no. 5, pp. 302–313, 2011.

[113] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *PVLDB*, vol. 8, no. 12, pp. 1466–1477, 2015.

[114] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, and M. Zaharia, "Weld: A common runtime for high performance data analytics," CIDR '17, 2017.

[115] A. Kumar, M. Boehm, and J. Yang, "Data management in machine learning: Challenges, techniques, and systems," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: ACM, 2017, pp. 1717–1722, ISBN: 978-1-4503-4197-4.

[116] L. N. Chakrapani, P. Korkmaz, B. E. Akgul, and K. V. Palem, "Probabilistic system-on-a-chip architectures," *in TODAES*, 2007.

[117] H. Cho, L. Leem, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *in TCAD*, 2012.

[118] V. Gupta *et al.*, "IMPACT: Imprecise adders for low-power approximate computing," in *ISLPED*, 2011.

[119] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *ICCAD*, 2013.

[120] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *DATE*, 2010.

[121] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI*, 2011.

[122] A. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012.

[123] D. Mohapatra *et al.*, "Design of voltage-scalable meta-functions for approximate computing," in *DATE*, 2011.

[124] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, 2004.

[125] S. Venkataramani *et al.*, "Salsa: Systematic logic synthesis of approximate circuits," in *DAC*, 2012.

[126] K. Nepal, Y. Li, R Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, 2014.

[127] Y. Liu *et al.*, "On logic synthesis for timing speculation," in *ICCAD*, 2012.

[128] A. Lingamneni *et al.*, "Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling," in *CF*, 2012.

[129] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013.

[130] S. Ramasubramanian *et al.*, "Relax-and-retime: A methodology for energy-efficient recovery based design," in *DAC*, 2013.

[131] S. Cheemalavagu, P. Korkmaz, K. V. Palem, B. E. S. Akgul, and L. N. Chakrapani, "A probabilistic cmos switch and its realization by exploiting noise," in *the Proceedings of the IFIP international*, 2005.

[132] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S. mo Kang, "An exact solution to the transistor sizing problem for cmos circuits using convex optimization," *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 1621–1634, 1993.

[133] N. Mohyuddin, P. Ehsan, and P. Massoud, *Probabilistic error propagation in a logic circuit using the boolean difference calculus*. Advanced Techniques in Logic Synthesis, Optimizations and Applications, 2011, pp. 359–381.

[134] A. Sampson *et al.*, "EnerJ: Approximate data types for safe and general low-power computation," *PLDI*, 2011.

[135] M. Carbin, S. Misailovic, and M. Rinard, "Verifying quantitative reliability of programs that execute on unreliable hardware," 2013.

[136] A. Rahimi, A. Marongiu, R. Gupta, and L. Benini, "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters," in *CODES+ISSS*, 2013.

[137] M. Kamal, A. Ghasemazar, A. Afzali-Kusha, and M. Pedram, "Improving efficiency of extensible processors by using approximate custom instructions," in *DATE*, Dresden, Germany, 2014, ISBN: 978-3-9815370-2-4.

[138] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ISCA*, 2010.

[139] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," *ISCA*, 2010.

[140] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ACM SIGARCH Computer Architecture News*, 2012.

[141] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *MICRO*, 2013.

[142] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.

[143] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 198–209, 2010.

[144] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, 2011.

[145] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014.

[146] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "AxGames: Towards crowdsourcing quality target determination in approximate computing," in *ASPLOS*, 2016.

[147] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[148] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *ISCA*, 2014.

[149] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for gpu throughput processors," in *MICRO*, Waikiki, Hawaii: ACM, 2015, pp. 482–493, ISBN: 978-1-4503-4034-2.

[150] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013.

[151] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *ASP-DAC*, 2014.

[152] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs," in *ISCA*, 2013.

[153] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *HPCA*, 2015.

[154] NetBSD Documentation, *How lazy FPU context switch works*, 2011.

[155] C. Clopper and E. S. Pearson, "The use of confidence or fiducial limits illustrated in the case of the binomial," *Biometrika*, pp. 404–413, 1934.

[156] M. H. DeGroot, *Probability and Statistics*. Chapman & Hall, 1974.

[157] B.-H. Lin, S.-H. Shieh, and C.-W. Wu, "A fast signature computation algorithm for lfsr and misr," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, no. 9, pp. 1031–1040, 2000.

[158] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, MIT Press, 1986.

[159] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012.

[160] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *DAC*, 2011.

[161] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and many-core architectures," in *MICRO*, 2009.

[162] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[163] S Galal and M Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 913–922, 2011.

[164] A. Sampson, P. Panchekha, T. Mytkowicz, K. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *PLDI*, 2014.

[165] M. Carbin, D. Kim, S. Misailovic, and M. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *PLDI*, 2012.

[166] B. Grigorian and G. Reinman, "Dynamically adaptive and reliable approximate computing using light-weight error analysis," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, IEEE, 2014, pp. 248–255.

[167] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *PLDI*, 2014.

[168] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," in *ASPLOS*, 2016.

[169] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ASPLOS*, 2015.

[170] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *ISCA*, Jun. 2015.

[171] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Prediction-based quality control for approximate accelerators," in *Workshop on Approximate Computing Across the System Stack (WACAS) in conjunction with ASPLOS*, Mar. 2015.

[172] J. Sacks, D. Mahajan, R. C. Lawson, and H. Esmaeilzadeh, "Robox: An end-to-end solution to accelerate autonomous control in robotics," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, California, Jun. 2018, ISBN: 978-1-5386-5984-7.

# VITA

Divya Mahajan was born in Chandigarh, India. She received her Bachelors in 2012 from the Indian Institute of Technology Ropar, where she was honored with the President of India Gold Medal for her outstanding academic performance. Subsequently, she completed her Masters in 2014 from the Department of Electrical and Computer Engineering at The University of Texas at Austin. She earned her PhD from the School of Computer Science at Georgia Institute of Technology, where she was advised by Professor Hadi Esmaeilzadeh as a part of the Alternate Computing Technologies lab. In recognition of her work, she was the recipient of the 2017 National Center for Women and IT (NCWIT) Collegiate Award and was awarded the distinguished paper award at HPCA 2016.

Divya's research spans multiple disciplines, ranging from Computer Architecture and Machine Learning to Database Management Systems. Central to her work is the concept that exploiting the full potential of hardware acceleration requires moving away from the limitations imposed by the traditional abstraction of the Instruction Set Architecture (ISA). Her research takes an alternative approach which delves into the algorithmic foundations of the application domain and devises novel abstractions between hardware and software. These abstractions then lay the foundation for the design of full compute stacks which automate the hardware acceleration process. While these guiding principles are very general, her work has particularly focused on how to apply them in the context of machine learning, database systems, and the intersection of these fields.

This dissertation was typeset in LaTeX.